O'REILLY®

TURING 图灵程序设计丛书

深入理解JavaScript特性

Practical Modern JavaScript

Dive into ES6 and the Future of JavaScript

[阿根廷] 尼古拉斯·贝瓦夸 著

李松峰 刘冰晶 高峰 译

黄小璐 欧雪雯 审校


中国工信出版集团


人民邮电出版社
POSTS & TELECOM PRESS

译者介绍

李松峰

360前端开发资深专家、前端TC委员、W3C AC代表，奇舞团Web字体服务“奇字库”作者。

刘冰晶

毕业于北京邮电大学，现为奇舞团前端开发工程师，专注于数据可视化以及前端动画领域。

高峰

硕士，毕业于中国科学技术大学软件学院。现为奇虎360前端开发工程师。

审校介绍

黄小璐

毕业于华中科技大学计算机学院。现为奇虎360软件开发工程师。参与翻译了《高性能HTML5》《移动Web手册》《大型JavaScript应用最佳实践指南》《Web开发权威指南》等书。

欧雪雯

前端开发工程师，多年从事科技、前端技术领域业余翻译工作。

数字版权声明


图灵程序设计丛书

深入理解JavaScript特性

Practical Modern JavaScript

[阿根廷] 尼古拉斯·贝瓦夸 著
李松峰　刘冰晶　高峰 译
黄小璐　欧雪雯 审校

Beijing • Boston • Farnham • Sebastopol • Tokyo O'REILLY®
O'Reilly Media, Inc.授权人民邮电出版社出版

人民邮电出版社
北　　京

图书在版编目（CIP）数据

深入理解JavaScript特性 /（阿根廷）尼古拉斯•贝瓦夸著 ; 李松峰，刘冰晶，高峰译. -- 北京 : 人民邮电出版社，2019.5
（图灵程序设计丛书）
ISBN 978-7-115-51040-2

Ⅰ. ①深… Ⅱ. ①尼… ②李… ③刘… ④高… Ⅲ. ①JAVA语言—程序设计 Ⅳ. ①TP312.8

中国版本图书馆CIP数据核字(2019)第059529号

内 容 提 要

本书旨在让读者轻松学习 JavaScript 的新进展，包括 ES6 及后续更新。书中提供了大量实用示例，以循序渐进的方式讲解了异步函数、对象解构、动态导入及异步生成器等内容。本书不仅介绍了箭头函数、解构、模板字面量以及其他语法方面的新元素，还全面展示了 ES6 引入的流程控制机制，以及如何高效地简化自己的代码。本书的讨论还涉及 ES6 内置的新集合类型、使用代理控制属性访问、ES6 中内置 API 的改进、CommonJS 与 ECMAScript 模块的互用性等方面。

本书适合前端开发者、编程爱好者和拥有 JavaScript 实际开发经验的程序员阅读参考。

◆ 著　　[阿根廷] 尼古拉斯·贝瓦夸
译　　李松峰　刘冰晶　高　峰
审　校　黄小璐　欧雪雯
责任编辑　温　雪
责任印制　周昇亮

◆ 人民邮电出版社出版发行　　北京市丰台区成寿寺路11号
邮编　100164　　电子邮件　315@ptpress.com.cn
网址　http://www.ptpress.com.cn
北京　　　　印刷

◆ 开本：800×1000　1/16
印张：16.25
字数：384千字　　　　2019年5月第1版
印数：1－3 500册　　　　2019年5月北京第1次印刷

著作权合同登记号　图字：01-2018-8087号

定价：79.00元

读者服务热线：(010)51095183转600　印装质量热线：(010)81055316

反盗版热线：(010)81055315

广告经营许可证：京东工商广登字 20170147 号

O'Reilly Media, Inc.介绍

O'Reilly Media 通过图书、杂志、在线服务、调查研究和会议等方式传播创新知识。自 1978 年开始，O'Reilly 一直都是前沿发展的见证者和推动者。超级极客们正在开创着未来，而我们关注真正重要的技术趋势——通过放大那些“细微的信号”来刺激社会对新科技的应用。作为技术社区中活跃的参与者，O'Reilly 的发展充满了对创新的倡导、创造和发扬光大。

O'Reilly 为软件开发人员带来革命性的“动物书”；创建第一个商业网站（GNN）；组织了影响深远的开放源代码峰会，以至于开源软件运动以此命名；创立了 *Make* 杂志，从而成为 DIY 革命的主要先锋；公司一如既往地通过多种形式缔结信息与人的纽带。O'Reilly 的会议和峰会集聚了众多超级极客和高瞻远瞩的商业领袖，共同描绘出开创新产业的革命性思想。作为技术人士获取信息的选择，O'Reilly 现在还将先锋专家的知识传递给普通的计算机用户。无论是通过书籍出版、在线服务还是面授课程，每一项 O'Reilly 的产品都反映了公司不可动摇的理念——信息是激发创新的力量。

业界评论

“O'Reilly Radar 博客有口皆碑。”

——*Wired*

“O'Reilly 凭借一系列非凡想法（真希望当初我也想到了）建立了数百万美元的业务。”

——*Business 2.0*

“O'Reilly Conference 是聚集关键思想领袖的绝对典范。”

——*CRN*

“一本 O'Reilly 的书就代表一个有用、有前途、需要学习的主题。”

——*Irish Times*

“Tim 是位特立独行的商人，他不光放眼于最长远、最广阔的视野，并且切实地按照 Yogi Berra 的建议去做了：‘如果你在路上遇到岔路口，走小路（岔路）。’回顾过去，Tim 似乎每一次都选择了小路，而且有几次都是一闪即逝的机会，尽管大路也不错。”

——*Linux Journal*

本书赞誉

“本书全面介绍了 ES6 新特性的语法和语义，有助于你大幅度提升代码的表达能力。作者把这些特性融入简单易懂的示例中，帮你快速上手。”

——Kent C. Dodds，PayPal 前端工程师，TC39 成员

“作者对重新定义现代 JavaScript 的诸多特性进行了深入彻底又贴合实际的解读。”

——Rod Vagg，require.io 技术顾问，Node.js TSC 成员

“原本复杂难懂的技术，被作者巧妙地转化为通俗易懂的文字和示例代码。”

——Mathias Bynens，Google 前端工程师，TC39 成员

“真正掌握 JavaScript 很难，2015 版规范又增加了很多新特性。本书化整为零，分别介绍了这些特性的使用场景、概念模型和最佳实践，而且示例很简单，因此掌握起来就容易多了。”

——Jordan Harband，Airbnb 软件工程师，TC39 成员

“ES6 为这门语言新增了很多特性，即使是富有经验的开发者也要花很多时间去学习掌握。在此过程中，我们需要一个向导，而本书是明智之选。”

——Ingvar Stepanyan，CloudFlare 前端工程师

“1995 年，我在发明 JavaScript 的时候，并没有想到它日后能成为互联网上使用最广的编程语言。本书与我循序渐进和直观明了的教学理念不谋而合。建议你好好读一读，从中发现对自己有用的东西，进而真正拥抱 JavaScript，最终致力于为所有人开发更好的 Web 应用。”

——Brendan Eich，JavaScript 之父

目录

序

1995 年，我在 Netscape 发明 JavaScript 的时候，并没有想到它日后能成为互联网上使用最广的编程语言。我只知道自己时间有限，要尽快写出一个“最小可发布的”版本。正因为如此，我将它写得尽量可扩展，自全局对象以下都可以修改，甚至连基础级的元对象协议钩子（如与 Java 方法同名的 `toString` 和 `valueOf`）也可以修改。

抛开其不断演进和越来越流行不说，JavaScript 始终受益于一种递进的、务实的教学方法，即“要事先行”。我认为这与匆忙设计和故意为之的可扩展性有着必然的联系。我重度使用了函数和对象这两个核心元素。因此，程序员可以将它们作为通用构造，花式地构建各种工具，最终造出一把锋利的“瑞士军刀”。而对学生来说，这意味着他们必须了解各种工具都能解决哪些任务，以及如何恰如其分地运用这些工具。

我感觉 Netscape 就像一阵旋风，相信 1995 年的那批同事也都有同感。那一年，Marc Andreessen 在路演时反复强调“Netscape+Java 会灭掉 Windows”，借与微软竞争而迅速实现了 IPO。Java 是正统编程语言、老大哥、“蝙蝠侠”，而 JavaScript 只是一个小兄弟、“助手罗宾”、小小的脚本语言。

但在编写第一版（代号 Mocha）时我就知道，JavaScript（而不是 Java）会与 Netscape 浏览器以及我同时创造的文档对象模型深度绑定。Netscape 与 Sun 或者浏览器与 JVM 的界限是无法跨越的，Java 只能作为插件嵌入。

因此我当时就有一种预感，JavaScript 要么会随时间的推移慢慢走向成功，要么会因为某些原因迅速消亡。记得我的好友兼室友 Jeff Weinstein 问我后 20 年干什么时，我说：“要么开发 JavaScript，要么完蛋。”那时，我就感觉好像欠了 JavaScript 用户一屁股债。这种感觉源于我在极短时间和管理层严令“看起来要像 Java”的双重限制下，选择了“双面瑞士军刀”这种设计。

“模块化 JavaScript”（Modular JavaScript Series）这套书与我循序渐进和直观明了的教学理念不谋而合，先从浅显易懂的代码示例讲起，逐步扩展到设计模式，再到完整的基于模块的应用构建。这套书匠心独运，专门探讨了有关测试的最佳实践和部署 JavaScript 应用的高超技术。说这套书是 O’Reilly 出版公司“JavaScript 图书”这顶王冠上又一颗璀璨的明珠一点也不为过。

看到尼古拉斯为此付出的努力，我感到非常欣慰，因为这本书一看就能给 JavaScript 程序员耳目一新的感觉。我第一次见到尼古拉斯是在巴黎的一次晚宴上，当时对他有了一点了解，之后就是线上交流。他写的东西特别实用，而且能够体谅 JavaScript 初学者，同时也非常幽默。这也促使我审阅了这本书的草稿。相信最终付梓时，拿到手的成品会更容易阅读，也更有意思。建议你好好读一读这本书，从中发现对自己有用的东西，进而真正拥抱 JavaScript，最终致力于为所有人开发更好的 Web 应用。

Brendan Eich

JavaScript 之父，Brave 软件公司 CEO 和联合创始人

前言

时间倒回 1998 年，那时我在学校用 FrontPage 做网页。如果当时有人告诉我，将来我会以 Web 开发为业，我一定会笑出声来。之后数年，JavaScript 一直向前发展，很难想象 Web 离开 JavaScript 是否还能像现在这样繁荣。本书将一点一点地勾勒出一幅完整的现代 JavaScript 图景。

读者对象

本书主要写给 Web（前端）开发者、编程爱好者和拥有 JavaScript 实际开发经验的程序员。这些人以及想要进一步了解 JavaScript 语言的任何人都可以通过阅读本书受益。

为什么编写本书

本书旨在让你轻松学习 JavaScript 的最新进展，包括 ES6 及后续更新。ES6 是这门语言一个里程碑式的版本，它几乎与简化的规范制定流程同时发布。当时我就围绕 ES6 的不同特性写了不少博文，受到了很多读者的喜爱。关于 ES6 的书也不少，但这些书与我心目中的关于 ES6 及其未来的书略有不同。本书在详细介绍 ES6 的新特性时，不会陷入规范、实现细节，也不会探讨那些几乎不可能遇到的边界情形（如果这些情况出现，肯定是要在网上查找相关资料的）。

本书不贪大求全，而是专注于循序渐进的学习过程，确保先学内容可以成为后学内容的基础，避免你为寻找某个定义翻来翻去。本书配备了大量实用示例，不仅涉及 ES6，还涉及 2015 年 6 月（ES6 规范定稿时间）以后的其他变化，包括异步函数、对象解构、动态导入、`Promise#finally` 和异步生成器。

本书的目标是确保你能够顺利继续学习本系列图书的后续其他分册。在这第一本书的基础上，我们将接着探讨模块化设计、测试和部署，到时就不必再细究代码示例中用到的语法

特性了。这种递进和模块化的学习方式体现在了整个系列中，包括每一本书、每一章，甚至每一节。

本书内容

第 1 章简单介绍了 JavaScript 及其标准化过程，回顾其发展，概述其现状，并展望其未来。这一章也会简单介绍 Babel 和 ESLint 这两个现代 JavaScript 开发必备的工具。

第 2 章介绍了 ES6 中最基础的部分，包括箭头函数、解构、`let` 和 `const`、模板字面量以及其他语法方面的新东西。

第 3 章讨论了用于声明对象原型的 `class` 语法、名为 `Symbol` 的新原始类型，以及 `Object` 对象的新方法。

第 4 章全面展示了 ES6 引入的流程控制机制。先从 Promise 开始，然后是迭代器、生成器和异步函数，这一章会详尽讨论每个主题并配有丰富的实例，揭示所有这些特性之间的协同关系。这一章不只是为了让你学会使用它们，更希望你能真正理解它们的最佳用途，从而简化自己的代码。

第 5 章介绍了 ES6 内置的新集合类型 `Set` 和 `Map`，`Set` 用于创建包含唯一值的集合，`Map` 用于创建对象映射。这一章也会给出实际的应用实例。

第 6 章展示了新的 `Proxy` 和 `Reflect` 内置特性，不仅会介绍如何使用代理，还会探讨为什么使用它们时要谨慎。

第 7 章主要讨论了 ES6 中的其他改进，特别是与 `Array`、`Math`、数值、字符串、Unicode 和正则表达式相关的改进。

第 8 章讨论了原生 JavaScript 模块，并简单介绍了其诞生的历史，最后详细介绍了其语法。

第 9 章的标题是“实用建议”，这与其他编程语言方面的图书有所不同。我没有将自己的这些建议放到各个章节，而是将它们汇编到一章。你将在这一章中看到什么时候该用哪种变量声明方式或字面量引号，异步代码流的控制，以及使用类和代理到底好不好之类的建议和思考。

如果你已经比较熟悉 ES6 了，那么建议你认真阅读第 4 章，其中关于流程控制的部分非常有价值。第 7 章和第 8 章也是必读的，因为其中的内容很难在别处看到。最后一章无疑会激发你的思考：在模块化 JavaScript 这个新领域中，什么样的做法合适，什么样的做法不可取？无论是否同意我的观点，你都会从中受益颇多。

排版约定

本书使用了下列排版约定。

- **黑体**
  表示新术语和重点强调的内容。

- 等宽字体（`constant width`）
  表示程序片段，以及正文中出现的程序元素，比如变量、函数名、数据库、数据类型、环境变量、语句和关键字等。



该图标表示提示或建议。



该图标表示警告或警示。

O’Reilly Safari

Safari（前身为 Safari Books Online）是一个会员制的培训和参考平台，面向企业、政府、教育从业者和个人。

Safari 用户可访问 O’Reilly Media、Harvard Business Review、Prentice Hall Professional、Addison-Wesley Professional、Microsoft Press、Sams、Que、Peachpit Press、Focal Press、Cisco Press、John Wiley & Sons、Syngress、Morgan Kaufmann、IBM Redbooks、Packt、Adobe Press、FT Press、Apress、Manning、New Riders、McGraw-Hill、Jones & Bartlett、Course Technology 等 250 多家出版社的上千种图书、培训视频、学习路径、交互式教程和精选播放列表。

如需了解更多信息，请访问 http://oreilly.com/safari。

联系我们

请把对本书的评价和问题发给出版社。

美国：

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472

中国：

北京市西城区西直门南大街 2 号成铭大厦 C 座 807 室（100035）
奥莱利技术咨询（北京）有限公司

O'Reilly 的每一本书都有专属网页，你可以在那儿找到本书的相关信息，包括勘误表、示例代码以及其他信息[1]。本书的网站地址是：http://www.oreilly.com/catalog/0636920047124。

对于本书的评论和技术性问题，请发送电子邮件到：
bookquestions@oreilly.com

要了解更多 O'Reilly 图书、培训课程、会议和新闻的信息，请访问以下网站：
http://www.oreilly.com

我们在 Facebook 的地址如下：

http://facebook.com/oreilly

请关注我们的 Twitter 动态：
http://twitter.com/oreillymedia

我们的 YouTube 视频地址如下：
http://www.youtube.com/oreillymedia

致谢

人类做的任何事都是以其他人的成果为基础的，本书也不例外。感谢 O'Reilly 公司的编辑。首先是 Nan Barber，她对我创作本书给予了极大的支持。还有 Ally MacDonald，她启发我以模块化方式来写这套书，而且帮助我克服了项目初期的困难。

本书的技术审校阵容可谓豪华。其中很多人都来自 TC39，也就是制定 JavaScript 标准的委员会，他们抽出宝贵的时间进行审阅，使本书的内容有了保证。Mathias Bynens（曾任职于 Opera）帮忙校对了本书中与 Unicode 标准相关的所有内容，确保了书中代码示例的高度一致。Kent C. Dodds（TC39 成员，PayPal）创造性地进行了视频评审，指出了一些问

注 1：读者可访问本书图灵社区页面（http://www.ituring.com.cn/book/2452）下载示例代码并提交中文版勘误。——编者注

题，进行了质量把关。Jordan Harband（TC39 成员，Airbnb）对书中讨论的很多 JavaScript 特性给出了技术意见。Alex Russell（TC39 成员，Google）帮助厘清了 JavaScript 的历史及有关标准主体的内容。Ingvar Stepanyan（Cloudflare）也是一位从代码中挑错的高手，帮我纠正了与规范底层细节相关的错误。Brian Terlson（TC39 编辑，Microsoft）也就 TC39 的时间安排和相关细节提供了解答。Rod Vagg（Node.js 项目）对本书代码示例的风格统一和更贴近主题提供了帮助。

Brendan Eich（TC39 成员，Brave CEO）分享了关于 JavaScript 早期和 TC39 的很多珍闻，这为第一章的内容打下了基础。如果没有他，也就不会有你手中这本书。

最后，我要感谢我的妻子 Marianela，感谢她在我撰写本书期间做出的牺牲和对我的宽容。Marian，没有你，我不可能完成本书！

电子书

扫描如下二维码，即可购买本书电子版。



第 1 章

ECMAScript和JavaScript的未来

JavaScript 已经从 1995 年的一个为了赢得战略优势的市场营销策略，变成了如今（2017 年）世界上使用最广泛的应用运行平台中的核心编程语言。该语言不再只是在浏览器中运行，现在也用于创建桌面和移动应用，还用于硬件设备，甚至是 NASA 的太空服设计。

JavaScript 是如何做到这一步的，接下来它又会怎么做呢？

1.1 JavaScript标准简史

1995 年，NetScape 公司想要构建一个动态的网页，但 HTML 无法实现这一点。为此，他们雇用 Brendan Eich 专门为浏览器开发一门功能类似于 Scheme 的语言。Brendan 加入之后，得知上级主管希望这门语言的语法像 Java，而且这一决定已经开始实施。

Brendan 花 10 天写出了 JavaScript 的第一个原型，主要实现了 Scheme 的一类函数和 Self 的原型等构造。这个初始版 JavaScript 的代号为 Mocha。它没有数组，没有对象字面量，任何错误都会给出警告。此外，它也没有异常处理，这也是如今仍有很多操作返回 `NaN` 或 `undefined` 的原因。Brendan 对 DOM0 级和 JavaScript 第一个版本的实现成为了这门语言标准化的基础。

1995 年 9 月，Netscape Navigator 2.0 beta 版发布，JavaScript 的修订版也内置其中，对外宣传时名叫 LiveScript。同年 12 月，Navigator 2.0 beta 3 发布，此时 LiveScript 又重新命名为 JavaScript（后成为 Sun 的注册商标，Sun 现在是 Oracle 旗下公司）。这次发布后不久，Netscape 公司推出了服务器端 JavaScript 的实现，用于在 Netscape Enterprise Server 上运行脚

本，并将其命名为 LiveWire。[1]1996 年，微软在 IE3 中推出 JScript，即通过逆向工程实现的 JavaScript。JScript 在服务器端也可以在互联网信息服务器（IIS，Internet information server）中运行。

1996 年，ECMA 的一个技术委员会 TC39 将 JavaScript 以名称 ECMAScript（ES）标准化为 ECMA-262 规范。为什么叫 ECMAScript 呢？因为 Sun 不同意将 JavaScript 商标转让给 ECMA，虽然微软提议叫 JScript，但其他成员公司又不想用，结果就只能使用 ECMAScript 这个尴尬的名字。

当时，TC39 开会主要就是争论该采用 Netscape 的 JavaScript，还是微软的 JScript。尽管如此，该委员会还是取得了成果，他们坚定不移地支持向后兼容，推动引入了严格相等运算符（`===` 和 `!==`），不会影响依赖松散相等比较算法的现有程序。

ECMA-262 的第一版于 1997 年 6 月发布。次年 6 月，国际标准化组织对这个规范进行了完善和认真审查，并以 ISO/IEC 16262 的形式发布，这就是它的第二版。

1999 年 12 月发布的第三版标准化了正则表达式、`switch` 语句、`do/whille`、`try/catch`、`Object#hasOwnProperty`，以及其他一些特性。其中大部分特性已经可以在 Netscape 的 JavaScript 运行环境 SpiderMonkey 中使用。

之后不久，TC39 发布了 ES4 规范的草案。ES4 的早期工作直接导致了 2000 年年中 JScript.NET 的产生[2]，并最终促进 2006 年 Flash 中 ActionScript 3 的诞生。[3]

此时，关于 JavaScript 应该朝哪个方向发展的不同意见导致了规范制定工作停滞不前。对于 Web 标准的发展而言，这时的情形很微妙：微软几乎垄断了 Web 行业，却对制定标准毫无兴趣。

2003 年，AOL 裁掉了 50 名 Netscape 员工[4]，Mozilla 基金会随之成立。同时，由于微软占据了超过 95% 的 Web 浏览器市场份额，TC39 被迫解散。

直到两年后，Brendan 在 Mozilla 以 Firefox 日益增长的市场份额为杠杆使得微软回归，ECMA 才得以重新开始 TC39 的相关工作。2005 年年中，TC39 再次开始召开定期会议。对于 ES4，计划引入模块系统、类、迭代器、生成器、解构、类型注释、尾调用优化、代数类型以及其他各类功能。这次的新增内容过于庞大，导致 ES4 一次又一次地延期了。

2007 年，TC39 分裂为两派：一派主张推出 ES3.1，即只在 ES3 的基础上完善改进；另一

注 1：1998 年的这个小册子（https://docs.oracle.com/cd/E19957-01/816-6411-10/contents.htm）详细介绍了服务器端 JavaScript 以及 LiveWire 的方方面面。

注 2：可以在微软网站上找到最初的公告（2000 年 7 月）。

注 3：Brendan Eich 在播客 JavaScript Jabber 中介绍了很多关于 JavaScript 起源的故事。

注 4：2003 年 7 月的 *The Mac Observer* 有该新闻的相关报道。

派则主张推出 ES4，新特性繁多，且亟待规范化。直到 2008 年 8 月[5]，两派才就 ES3.1 达成一致，ES3.1 后来又发展为 ES5。ES4 的提议似乎废弃了，但实际上其中很多特性最终写进了 ES6（达成和解时美其名曰 Harmony，即“和谐”），当然，其中一些特性还在讨论，另外少数特性确实已经废弃、被拒或撤销。ES3.1 也为 ES4 的逐步实现奠定了基础。

2009 年 12 月，ES3 发布 10 周年，ECMAScript 第五版发布。这一版汇集了当时浏览器中业已存在的扩展，或者说“事实标准”，增加了 `get` 和 `set` 存取器，增强了 `Array` 原型的函数特性，还引入了反射和内省机制，以及对 JSON 解析和严格模式的支持。

2011 年 6 月，经过再次审查和编辑，该规范形成了第三版的国际标准 ISO/IEC 16262:2011，并作为 ECMAScript 5.1 发布。

2015 年 6 月，TC39 又花 4 年完成了 ECMAScript 6。第六版是该语言面世以来改动最大的一个版本，实现了许多 ES4 中被推迟到 Harmony 中的提案。本书主要探讨的就是 ES6。

在 ES6 的制定过程中，另一个旨在推动 Web 发展的组织 WHATWG（网页超文本应用技术工作小组）于 2012 年推出了一个文档，以记录 ES5.1 和浏览器实现在兼容性和可操作性方面的差异。这个工作小组标准化了之前规范中没有提及的 `String#substr`，统一了在 HTML 标签中包装字符串的几种方法，明确写出了 `Object.prototype` 中的 `__proto__` 和 `__defineGeter__` 等原型属性，同时还做出了其他一些改进。[6] 这些工作最终形成了一份独立的 Web ECMAScript 规范，最终于 2015 年被加入到附录 B 中。附录 B 是 ECMAScript 核心规范中的参考部分，这意味着浏览器可以不遵照其进行实现。此次更新之后，附录 B 也成为了 Web 浏览器的规范和要求。

第六版是 JavaScript 历史上一个意义重大的里程碑。除了众多新功能之外，ES6 也是 ECMAScript 成为持续迭代标准的一个转折点。

1.2 持续迭代的ECMAScript

ES3 在 10 年内未有重大改变，之后 4 年才推出 ES6。这一漫长历程清楚地表明 TC39 流程需要改进。之前的修订流程是最终发布日期驱动的。只要有议题未能达成一致，下次修订就会变得遥遥无期。时间一长又会有新的特性提出，进而导致更多拖延。小的修订总会因大的新增而拖延，而大的新增迫于最终发布日期的压力，就会仓促通过修订，以免造成一拖再拖。

ES6 发布后，TC39 优化了提案修订的流程[7]，以满足现代预期：迭代具有经常性和一贯性，且规范的制定要更加民主化。基于这一点，TC39 不再采用古老的 Word 文档形式，而是使

注 5：2008 年，Brendan Eich 给 es-discuss 发了一封邮件，其中介绍了当时的情况，此时距 ES3 发布都快 10 年了。
注 6：要想了解将 Web ECMAScript 规范合并到主干时所做的全部更改，请参见 WHATWG 博客。
注 7：2013 年 9 月的 PPT，*Post-ES6 Spec Process*，介绍了优化后的提案修订流程。

用 Ecmarkup（用于编写 ECMAScript 规范的 HTML 语言的超集）和 GitHub 来提交需求，大大增加了非成员的提案数量 [8]，他们就像是外在的参与者一样。这种新形式是持续的，并且更加透明：最新的规范草案随时可以查看，而无须像之前那样，必须从网页下载 Word 文档或 PDF 版本。

Firefox、Chrome、Edge、Safari 以及 Node.js 对 ES6 规范的支持均已超过 95%[9]。现在我们就可以在这些浏览器中使用已经支持的功能，而不用等到它们对 ES6 的支持度达到 100%。

新的流程引入了 4 个不同的成熟度阶段。[10] 提案越成熟，最终添加到规范中的可能性越大。

只要还未作为正式提案提交，关于修改或新增内容的任何讨论、想法或者提议都会被视为有前途的“稻草人”提案（阶段 0），但只有 TC39 委员会的成员才可以创建“稻草人”提案。在编写本书时，有 10 多个活跃的“稻草人”提案。[11]

阶段 1 表示提案被正式提出，希望能够解决各方关注的问题，厘清与其他提案的交集，并实现问题。这一阶段的提案需要明确描述一个具体的问题，并给出该问题的具体解决方案。阶段 1 的提案通常包括以下几方面的内容：高级 API 描述、演示性的用法示例、内部语义和算法的讨论。阶段 1 的提案可能会随着流程的推进而发生很大的改变。

阶段 2 的提案是规范的初步草案。从这一步开始，需要在运行环境中验证具体的实现。实现的方式可以是腻子脚本（polyfill）、能让运行环境支持提案的用户代码、原生支持提案内容的引擎实现，也可以是使用构建工具将源代码转换、编译成现有引擎可以执行的代码。

阶段 3 的提案是候选推荐提案。只有规范的编辑和指定的审查人员在最终的规范上签字确认，提案才能进入阶段 3。另外，还需要实现者表示出对该提案感兴趣。实际上，只有满足以下 3 个条件之一，提案才能进入阶段 3：某个浏览器已经实现该提案，有高度吻合的腻子脚本，有类似 Babel 的实时编译工具的支持。除了修复使用过程中新发现的问题，阶段 3 的提案不会再有其他改动。

要想进入阶段 4，提案必须有两个独立的实现方案通过验收测试。进入阶段 4 的提案最终会添加到 ECMAScript 的下一版中。

从现在开始，ECMAScript 预计每年都会发布新版本。为了与年度发版计划统一，规范的版本从现在开始与出版的年份相关联。因此，ES6 也就是 ES2015，之后将有 ES2016（而不是 ES7）、ES2017，等等。实际上，ES2015 这个称呼并没有被接受，大家还是习惯称其为 ES6。ES2016 也是在命名约定改变前就发布了的，因此有时人们也称其为 ES7。由于

注 8：TC39 采纳的提案参见 https://mjavascript.com/out/tc39-proposals。

注 9：ES6 的浏览器兼容性报告参见 https://kangax.github.io/compat-table/es6/。

注 10：TC39 的提案流程文档参见 https://tc39.github.io/process-document/。

注 11：“稻草人”提案参见 https://github.com/tc39/proposals/blob/master/stage-0-proposals.md。

ES6 这一名称已经为社区普遍接受，我们抛开不谈。最终的规范版本将是 ES6、ES2016、ES2017、ES2018，以此类推。

调整后的提案流程加上每年都发布一版的强制约定形成了更加一致的发布过程。这也意味着规范的修订版本号变得不再那么重要。现在的重点是提案的不同阶段，我们相信将来大家会越来越少提及 ECMAScript 标准的某个特定修订版。

1.3 浏览器支持和辅助工具

如果能够有两个 JavaScript 引擎提供独立的实现，阶段 3 的候选推荐提案最有可能进入下一个版本的规范之中。实际上，只要阶段 3 的提案有实验性的引擎实现或者腻子脚本，再或者可以通过编译器得到支持，那么就已经能够安全地在实际开发中使用了。其实，阶段 2 和更早阶段的提案也有为 JavaScript 开发人员所使用的，这也加强了实现者和使用者之间的反馈循环。

Babel 等将代码作为输入并生成 Web 平台原生支持的输出（HTML、CSS 或 JavaScript）的编译器通常称为**转译器**（transpiler），它是编译器的一个子集。如果我们想要在代码中使用某个 JavaScript 引擎还没有普遍实现的提案，Babel 之类的编译器可以帮我们将相应的代码转换成现有 JavaScript 实现可以运行的代码。

代码转换是在构建时完成的，因此用户拿到的是自己的 JavaScript 运行环境所支持的代码。这一机制降低了对运行环境的要求，也让 JavaScript 开发者能够更早地使用新的语言功能和语法。对于规范的编写者和实现者来说，这也是非常有利的，因为这样他们就可以得到可行性、迫切性、可能存在的 bug，以及边界用例等方面的反馈。

转译器可以将 ES6 代码转换成浏览器普遍可以解释的 ES5 代码。这是如今在生产环境中运行 ES6 代码的最可靠方式：通过构建生成 ES5 代码，这样新旧浏览器都可以执行。

这种方式同样适用于 ES7 及后续版本。由于语言规范每年都会发布新版本，我们可以期待编译器支持 ES2017 输入、ES2018 输入，等等。同样，随着浏览器的支持度越来越好，编译器可以逐渐降低支持 ES6 输出、ES7 输出的复杂性。从这种意义上说，我们可以将 JavaScript 转译器看作移动的窗口，其输入是使用最新语法编写的代码，输出是不影响浏览器运行的最新代码。

接下来我们探讨如何在工作中使用 Babel。

1.3.1 Babel转译器简介

Babel 可以将 ES6 代码编译成 ES5 代码。生成的 ES5 代码很容易看懂，因此非常适合还不完全熟悉新特性的人来理解新特性。

Babel 的在线读取－求值－打印循环（REPL，read-evaluate-print loop）转换器是学习 ES6 的好帮手，无须安装 Node.js、babel CLI，也无须手工编译源码。

REPL 提供了一个源码输入框，用于自动实时编译，编译后的代码位于源码右侧。

我们可以在 REPL 中写几行代码，如下所示。

```
var double = value => value * 2
console.log(double(3))
// <- 6
```

我们可以在右侧看到转换后的 ES5 等价代码，如图 1-1 所示。更新源码时，转译结果也会实时更新。

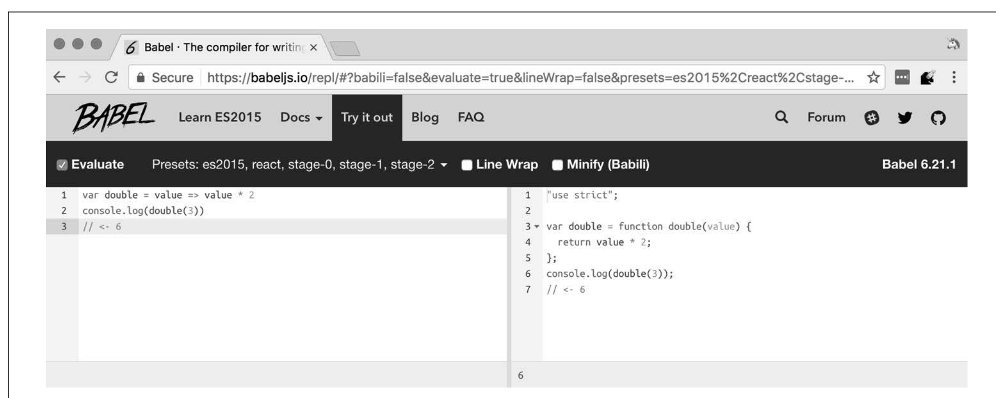


图 1-1：在线 Babel REPL 是交互式学习 ES6 的好工具

Babel REPL 也是用于试验本书中介绍的一些特性的有效工具。但需要知道的是，Babel 并不转换新的内置对象，如 `Symbol`、`Proxy` 和 `WeakMap`。这些引用会原封不动地保留下来，最终由执行 Babel 输出代码的运行环境提供这些内置对象。如果想要支持还没有实现这些内置对象的运行环境，可以在代码中引入 `babel-polyfill` 包。

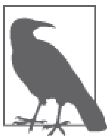
在一些较老的 JavaScript 版本中，想要语义正确地实现某些特性是很难的，甚至完全不可能。此时可以用腻子脚本弥补这个问题，但腻子脚本通常不能覆盖所有情况，因此需要做一些妥协。所以，在将转译后的使用了内置对象和腻子脚本的代码发布到生产环境之前，最好多做一些测试。

考虑到这种情况，最好还是等浏览器整体支持这些新的内置对象时再使用它们。我们建议你使用不依赖于这些新内置对象的替代方案。与此同时，学习这些特性也很重要，这样我们对 JavaScript 语言的理解才不会落后。

Chrome、Firefox 和 Edge 等现代浏览器现在已经支持 ES2015 及后续版本的大部分内容，所以在支持的前提下，可以通过它们的开发者工具来尝试新特性。产品级应用需要

依赖新 JavaScript 特性时，还是推荐使用转换器进行编译处理，这样应用能够支持更多的 JavaScript 运行环境。

除了 REPL，Babel 还提供了一个在命令行中运行的 Node.js 包，可以通过 Node.js 的包管理工具 `npm` 来安装它。



下载 Node.js。安装 `node` 后，就可以在终端内使用 `npm` 命令了。

开始之前，我们先创建一个项目目录以及一个用于描述 Node.js 应用的 package.json 文件。可以通过 `npm` 在命令行中创建 package.json 文件。

```
mkdir babel-setup
cd babel-setup
npm init --yes
```



执行 `init` 命令时传递 `--yes` 参数表示不用询问我们，可以使用 `npm` 提供的默认值来配置 package.json 文件。

再创建一个 example.js 文件，并在其中加入以下代码，然后将其保存到刚刚创建的 babel-setup 目录下的 src 子目录。

```
var double = value => value * 2
console.log(double(3))
// <- 6
```

在常用的终端中输入以下两行命令即可安装 Babel。

```
npm install babel-cli@6 --save-dev
npm install babel-preset-env@6 --save-dev
```



通过 `npm` 安装的包存放于项目根目录的 node_modules 目录下。可以通过创建 `npm script` 命令或使用 `require` 声明来访问这些包。

`--save-dev` 参数会将所安装的包作为开发依赖添加到 package.json 文件中。这样一来，当我们将项目移植到一个新环境时，只须运行 `npm install` 命令就可以重新安装每个依赖。

`@` 符号指明了包的特定版本。使用 `@6` 则是告诉 `npm` 安装 `babel-cli` 的 `6.x` 版本中最新的一个。这种偏好限制可以确保我们的应用将来不出问题，因为它可以确保永远不会安装 `7.0.0` 或后续版本，从而避免了 `7.0.0` 之后的版本包含当前无法预见的破坏性变化。

接下来，我们修改 package.json 中的 `scripts` 属性的值，如下所示。`babel-cli` 提供的 `babel` 命令行工具可以获取 src 目录下的全部内容，将它们编译成目标输出格式，并将结果保存到 dist 目录中，并且保留文件的原始目录结构。

```
{
  "scripts": {
    "build": "babel src --out-dir dist"
  }
}
```

加上前面安装的包，现在我们构成了以下这个超小的 package.json 文件。

```
{
  "scripts": {
    "build": "babel src --out-dir dist"
  },
  "devDependencies": {
    "babel-cli": "^6.24.0",
    "babel-preset-env": "^1.2.1"
  }
}
```



`scripts` 对象中列举的所有命令都可以通过 `npm run <name>` 执行，这种执行方式会临时修改环境变量 `$PATH` 的值，这样我们才可以在系统未全局安装 `babel-cli` 的情况下找到 `babel-cli` 并在命令行执行。

如果现在在终端内执行 `npm run build`，你就能看到生成后的 dist/example.js 文件。输出的文件和源文件完全相同。这是因为 Babel 还不知道编译的目标格式，我们需要先配置它。在 package.json 旁创建一个 .babelrc 文件，并在其中写入以下 JSON。

```
{
  "presets": ["env"]
}
```

这里的 `env` 就是前面通过 `npm` 安装的 `babel-preset-env`，它给 Babel 添加了一系列插件，以便将不同的 ES6 代码转换为 ES5。这个预设包含很多插件，其中就有插件将 example.js 中的箭头函数转换成 ES5 代码。`env` 预设会根据最新浏览器已经支持的特性启用相关的转换插件。这个预设是可配置的，我们可以决定向后兼容到哪个浏览器。兼容的浏览器越多，编译生成的包就越大；兼容的浏览器越少，能满足的用户就越少。当然，到底什么配置最合适，最终还是要经过研究才能决定。默认配置是启用所有转换插件，兼容尽可能多的运行环境。

再运行一次编译脚本就能够看到输出的 ES5 代码了。

```
» npm run build
» cat dist/example.js
"use strict"
```

```
var double = function double(value) {
  return value * 2
}
console.log(double(3))
// <- 6
```

接下来我们再介绍一个代码检查工具 `eslint`，它可以帮助我们确保项目代码的质量。

1.3.2　使用ESLint提高代码质量和一致性

开发项目时，我们会发现冗余或无用的代码，编写很多新代码，删除无关或没必要的功能，也会为适应新架构而来回搬运代码。随着代码越写越多，团队规模也会随之变化。刚开始可能只有几个人，甚至只有一个人，但随着项目规模越来越大，参与编码的人也会越来越多。

代码检查工具可以帮助我们发现语法错误。现代检查工具通常都是可定制的，允许我们建立适合自己团队的代码约定。坚守一致的代码风格和质量基准可以使团队的编码风格趋于一致。不同的团队成员可能会对代码风格有不同的意见，但有了代码检查工具和协商一致的配置后，这些意见就变成了明确可遵循的规则。

首先是确保程序能被解析，其次可能要防止 `throw` 抛出字符串字面量作为异常，或者不允许在生产环境中使用 `console.log` 和 `debugger` 语句。然而，要求所有函数调用都只能有一个参数就有点过了。

虽然代码检查工具能够定义并强制推行一种编码风格，但定义规则时也不能太任性。如果规则太严格，可能会影响开发效率，得不偿失。反之，规则太宽松的话，代码就不能保持统一风格。

要想把握好这个度，就应该尽量不使用多数情况下都不能改善程序的规则。新增一条规则时，应当扪心自问，添加它能否显著改善现有代码库以及未来的新代码？

ESLint 是一个现代代码检查工具，它集合了一些插件，具有不同的规则，支持自定义。我们可以决定不遵守规则时是输出警告还是导致停机错误。与安装 `babel` 一样，我们也可以通过 `npm` 安装 `eslint`。

```
npm install eslint@3 --save-dev
```

接下来我们需要配置 ESLint。因为本地安装了 `eslint`，所以可以在 node_modules/.bin 中找到它的命令行工具。执行以下命令能够引导我们配置 ESlint。首先，告诉它我们想使用一套流行的风格，然后选择 Standard[12]，最后选择 JSON 格式的配置文件。

注 12：注意，Standard 是一种自我宣告，并未由任何官方组织进行实际的标准化。其实，只要能够保持统一，使用哪种风格并不重要。在阅读项目代码时，一致性有助于减少困扰。Airbnb 风格指南也是很受欢迎的，与 Standard 不同，它默认不可省略分号。

```
./node_modules/.bin/eslint --init
? How would you like to configure ESLint?
  Use a popular style guide
? Which style guide do you want to follow? Standard
? What format do you want your config file to be in? JSON
```

除了个别规则，eslint 还支持扩展预定义规则，这些规则以 Node.js 包的形式存在。在多项目甚至社区中共享配置时，这会很方便。选择 Standard 后，可以看到 ESLint 向 package.json 中添加了一些依赖，即包含预设 Standard 规则的包，同时创建了一个名为 .eslintrc.json 的文件，其中包含以下内容。

```
{
  "extends": "standard",
  "plugins": [
    "standard",
    "promise"
  ]
}
```

直接引用包含 npm 实现细节的 node_modules/.bin 目录并不好。虽然前面初始化 ESLint 配置时这么引用了，但其实应该避免这样做，而且检查代码时也不应该每次都那么输入一遍。为此，我们在 packge.json 中添加一个脚本命令。

```
{
  "scripts": {
    "lint": "eslint ."
  }
}
```

前面安装 Babel 时提到过，npm run 在执行脚本命令时会将 node_modules 加入 PATH 环境变量。这样一来，在检查代码时，只要执行 npm run lint，npm 就会在 node_modules 目录中找到 ESLint CLI。

我们来看看以下的示例文件 example.js，其中的代码故意没有遵守规则，以演示 ESLint 具体做了什么。

```
var goodbye='Goodbye!'

function hello(){
  return goodbye}

if(false){}
```

执行 lint 脚本命令时，ESLint 会标识文件中所有错误的地方，如图 1-2 所示。

```
bevacqua@MacBook-Pro: ~/dev/practical-es6/code/ch01/ex02-eslint-setup

» npm run lint

> @ lint /Users/bevacqua/dev/practical-es6/code/ch01/ex02-eslint-setup
> eslint .


/Users/bevacqua/dev/practical-es6/code/ch01/ex02-eslint-setup/src/example.js
  1:12  error  Infix operators must be spaced                                                                   space-infix-ops
  1:23  error  Extra semicolon                                                                                  semi
  3:10  error  'hello' is defined but never used                                                                no-unused-vars
  3:15  error  Missing space before function parentheses                                                        space-before-function-paren
  3:17  error  Missing space before opening brace                                                               space-before-blocks
  4:3   error  Closing curly brace should be on the same line as opening curly brace or on the line after the previous block  brace-style
  4:17  error  Requires a space before '}'                                                                      block-spacing

✖ 7 problems (7 errors, 0 warnings)
```

图 1-2：ESLint 可以帮助我们检查语法错误，保持代码风格统一

如果在执行命令时传递 `--fix` 参数，那么 ESLint 能够自动修复大多数的风格问题。在 package.json 中加入以下脚本命令。

```
{
  "scripts": {
    "lint-fix": "eslint . --fix"
  }
}
```

此时执行 `lint-fix` 只会看到两个错误：未使用 `hello` 以及 `false` 是一个不变的条件。其他错误都已经修复了，结果如以下代码所示。上述两个错误没有修复，因为 ESLint 会保持中立，不对代码语义做预设推断，以免导致语义改变。这样一来，`--fix` 就能帮助我们有效解决编码风格问题，同时又不会破坏逻辑。

```
var goodbye = 'Goodbye!'

function hello() {
  return goodbye
}

if (false) {}
```



`prettier` 也是一种代码检查工具，可用于自动格式化代码。我们可以配置 `prettier` 自动重写代码，以确保代码遵循设定的首选项，如使用给定的空格缩进，统一使用单引号或双引号，末尾自动加逗号，或者限制最大行长度。

现在我们知道了如何将现代 Javascript 代码编译成每个浏览器都能理解的代码，以及如何正确检查和格式化代码。接下来概述一下 ES6 的特性，并展望一下 JavaScript 的未来。

1.4　ES6特性

ES6 规范足足有 566 页，是 ES5.1 规范 258 页的两倍多。ES6 的主要变化可以归纳为以下几类：

- 语法糖
- 新机制
- 更好的语义
- 更多的内置对象和方法
- 对原有限制的非破坏性解决方案

语法糖是 ES6 中最重要的组成部分，包括用新类来表达对象继承、箭头函数以及属性值简写等一系列更简洁的语法。此外，解构、剩余参数以及扩展运算符等我们将介绍的特性也提供了更加语义化的编程方式。第 2 章和第 3 章将介绍 ES6 中的这部分内容。

ES6 提供了几种新机制来描述异步流程：代表一个操作最终结果的 Promise、代表一系列值的**迭代器**，以及能产生一系列值的特殊迭代器——**生成器**。基于这些新概念和结构，ES2017 提供了 `async/await`，让我们可以像编写同步代码那样编写异步代码。第 4 章将介绍这里提到的迭代及流控制机制。

在 JavaScript 中，使用任意字符串作为键的普通对象来创建映射是很常见的。如果键值来自用户输入，且没有验证，那么极有可能产生漏洞。为此，ES6 引入了一些新的原生内置对象来管理集合和映射，并且没有只能使用字符串作为键的限制。第 9 章将介绍相关内容。

代理对象用于重新定义可以通过 JavaScript 反射完成的操作。代理对象和其他语境中的代理类似，比如网络路由中所说的代理。它可以拦截 JavaScript 对象的任何交互，比如属性的定义、删除以及访问。鉴于代理的工作机制，我们很难用腻子脚本全面实现其功能：当然，腻子脚本是有的，只是在某些场景下其实现与规范不符。第 6 章将介绍代理。

除了新的内置对象，ES6 还对 `Number`、`Math`、`Array` 以及 `String` 对象进行了一定的扩展。第 7 章将介绍添加在这些内置对象上的一系列新实例方法和静态方法。

ES6 还为 JavaScript 引入了一个原生模块系统。第 8 章从 Node.js 使用的 CommonJS 模块系统讲起，然后详细讲解 JavaScript 原生模块的语义。

因为 ES6 引入了太多修改，所以很难做到将其新特性与现有 JavaScript 知识自然整合。为此，第 9 章将用一整章的篇幅来分析每个特性的优点和重要性，以便你对 ES6 建立起初步的概念，并基于此开始使用 ES6。

1.5 JavaScript的未来

JavaScript 语言已经从 1995 年一门没啥名气的语言发展为今天这样一门强大的语言。虽然 ES6 向前跨越了一大步，却远远未到终点。鉴于每年都会有新的规范发布，如何跟上新规范发布的步伐就很重要了。

了解 1.2 节中介绍的持续迭代流程后，我们知道，要想跟进标准，首先就要定期访问 TC39 提案库[13]。我们要时刻关注候选推荐提案（即阶段 3 的提案），因为这些提案最有可能加入新规范。

用一本书来介绍一门快速发展的语言是不太可能的。因此，关注 TC39 提案库、订阅周刊[14]、阅读 JavaScript 博客[15] 是及时跟进 JavaScript 最新进展的有效方式。

撰写本书时，开发者期待已久的 Async 函数已经加入规范，并在 ES2017 中发布。此时此刻有很多候选提案，比如支持异步加载原生 JavaScript 模块的动态 `import()`，以及使用 ES6 中针对参数列表和数组引入的剩余和扩展运算符来枚举对象属性。

虽然本书的主要关注点是 ES6，但我们同样会学习重要的候选推荐，如刚刚提及的 Async 函数、动态 `import()` 调用、对象剩余 / 扩展，以及其他内容。

注 13：TC39 收录的所有提案参见 https://mjavascript.com/out/tc39-proposals。

注 14：这种电子周刊很多，如 *Pony Foo Weekly*、*Javascript Weekly*。

注 15：Pony Foo 网站上有很多关于 ECMAScript 开发的文章，Axel Rauschmayer 也写了很多关于这方面的文章。

第 2 章

ES6基础

ES6 引入了大量的非破坏性语法改进，本章将逐一讨论其中大部分改进。其中很多是语法糖，因为使用更复杂的 ES5 代码也能够实现。当然，也有一部分不止是语法糖，如 `let` 和 `const` 这两种完全不同的变量声明方式，本章最后会对其进行介绍。

ES6 中的对象字面量语法有一些调整，我们从这部分开始说起。

2.1　对象字面量

对象字面量是指使用 {} 简写语法进行的对象声明，具体的语法格式如下所示。

```
var book = {
  title: 'Modular ES6',
  author: 'Nicolas',
  publisher: 'O´Reilly'
}
```

ES6 对对象字面量语法进行了一些小的改进：属性值简写、可计算属性名和方法定义。接下来我们将探讨这些内容及其用法。

2.1.1　属性值简写

有时我们会声明这样一个对象，该对象的属性名和所引用的变量名相同。假设有一个 `listeners` 数组，要想将其赋给对象字面量中名为 `listeners` 的属性，必须重复输入该名称。以下代码中的对象字面量就包含两个名称与值重复的属性。

```
var listeners = []
function listen() {}
var events = {
  listeners: listeners,
  listen: listen
}
```

在 ES6 中，借用新的属性值简写语法可以省略属性值和冒号。属性值简写会进行隐性赋值，如下所示。

```
var listeners = []
function listen() {}
var events = { listeners, listen }
```

随着本章的讨论，我们会发现属性值简写可以在不影响代码含义的情况下有效减少重复代码。以下代码重新实现了浏览器中的持久存储 API `localStorage` 的部分内容，可以将其看作内存中的 ponyfill[1]。如果没有使用简写语法，那么 `storage` 对象看起来会更加冗长。

```
var store = {}
var storage = { getItem,  setItem, clear }
function getItem(key) {
  return key in store ? store[key] : null
}
function setItem(key, value) {
  store[key] = value
}
function clear() {
  store = {}
}
```

ES6 中的很多特性旨在减少所维护代码的复杂性，属性值简写只是其中之一。习惯这一语法后，你会发现代码的可读性和开发者的生产效率都会得到提升。

2.1.2 可计算属性名

有时我们可能需要声明一个属性名依赖变量或表达式的对象，如以下的 ES5 代码所示。在这个示例中，假设 `expertise` 是一个函数参数，且事先并不知道它的值。

```
var expertise = 'journalism'
var person = {
  name: 'Sharon',
  age: 27
}
```

注 1：与 polyfill 一样，ponyfill 是对尚未被所有 JavaScript 运行环境所支持的功能的用户实现。polyfill 主要试图修补运行环境，从而使某一特性看起来像原生支持的，而 ponyfill 是将运行环境所缺失的功能实现为一个独立的模块,并且不会污染运行环境。这样做的好处是不会超出第三方库对运行环境的期望，因为它们可能并不知道我们所使用的 polyfill。

```
person[expertise] = {
  years: 5,
  interests: ['international', 'politics', 'internet']
}
```

ES6 中的对象字面量不再局限于使用静态名称声明属性。我们也可以使用可计算属性名，用方括号包裹表达式，并将其作为属性名使用。执行声明时，表达式才会被计算，计算结果将作为属性名使用。以下示例只用一步就完成了上例 `person` 对象的声明，无须借助第二个声明来添加 `expertise` 属性。

```
var expertise = 'journalism'
var person = {
  name: 'Sharon',
  age: 27,
  [expertise]: {
    years: 5,
    interests: ['international', 'politics', 'internet']
  }
}
```

属性值简写和可计算属性名不能同时使用。属性值简写是编译时执行的简单语法糖，可以帮助避免重复，可计算属性名则是运行时计算的。如果试图同时使用这两个不兼容的特性，那么系统会抛出语法错误异常。大多数情况下，这种组合会导致代码难以理解，因此最好不要同时使用它们。

```
var expertise = 'journalism'
var journalism = {
  years: 5,
  interests: ['international', 'politics', 'internet']
}
var person = {
  name: 'Sharon',
  age: 27,
  [expertise] // 语法错误
}
```

可计算属性名的常见使用场景是，当我们想要将 `grocery` 添加到另一个对象中，并且想要使用 `grocery.id` 字段作为键值时，就可以使用可计算属性名，如下所示。我们可以直接在 `groceries` 对象字面量中进行内联声明，无须额外使用第三条声明语句将 `grocery` 添加到 `groceries` 中。

```
var grocery = {
  id: 'bananas',
  name: 'Bananas',
  units: 6,
  price: 10,
  currency: 'USD'
}
var groceries = {
```

```
  [grocery.id]: grocery
}
```

另一种使用场景是，一个函数接受一个参数，然后用该参数构建对象。如果使用 ES5，则需要分配一个变量来声明对象字面量，然后添加一个动态属性，最后返回这个对象。以下示例展示了这一使用场景，创建了一个能够用于 Ajax 消息的 envelope，这些消息遵循以下约定：具有一个描述发生的错误的 `error` 属性，以及一个表示成功的 `success` 属性。

```
function getEnvelope(type, description) {
  var envelope = {
    data: {}
  }
  envelope[type] = description
  return envelope
}
```

可计算属性名只需要一条语句即可实现以上函数。

```
function getEnvelope(type, description) {
  return {
    data: {},
    [type]: description
  }
}
```

接下来我们要介绍的有关对象字面量的改进是关于方法的。

2.1.3 方法定义

通常来说，我们可以通过添加属性为对象声明方法。以下代码创建了一个能够支持多种事件类型的小型事件分发器。`emitter#on` 方法可以用于注册事件监听函数，`emitter#emit` 方法可以用于分发事件。

```
var emitter = {
  events: {},
  on: function (type, fn) {
    if (this.events[type] === undefined) {
      this.events[type] = []
    }
    this.events[type].push(fn)
  },
  emit: function (type, event) {
    if (this.events[type] === undefined) {
      return
    }
    this.events[type].forEach(function (fn) {
      fn(event)
    })
  }
}
```

从 ES6 开始，可以在对象字面量中使用新的方法定义语法进行方法声明，省略冒号和 function 关键字。与使用 function 关键字的传统声明方式相比，这样更加简洁。以下示例展示了使用方法定义时的 emitter 对象，如下所示。

```
var emitter = {
  events: {},
  on(type, fn) {
    if (this.events[type] === undefined) {
      this.events[type] = []
    }
    this.events[type].push(fn)
  },
  emit(type, event) {
    if (this.events[type] === undefined) {
      return
    }
    this.events[type].forEach(function (fn) {
      fn(event)
    })
  }
}
```

箭头函数是 ES6 中的另一种函数声明方式，它有好几种形式。接下来我们探究一下箭头函数到底是什么、如何声明，并探讨其语法形式。

2.2 箭头函数

在 JavaScript 中，我们通常使用如下代码进行函数声明，其中包括函数名、参数列表和函数体。

```
function name(parameters) {
  // 函数体
}
```

我们也可以创建匿名函数，省略函数的名称，将其赋给一个变量或对象的属性，又或者直接调用。

```
var example = function (parameters) {
  // 函数体
}
```

从 ES6 开始，可以将箭头函数作为声明匿名函数的另一种方式。需要注意的是，箭头函数有几种不同的写法。以下代码中的箭头函数和前面所见的匿名函数非常相似。不同之处在于缺少了 function 关键字，并且参数列表的右侧多了一个箭头 =>。

```
var example = (parameters) => {
  // 函数体
}
```

虽然箭头函数看起来和常规的匿名函数很相似，但它们本质上完全不同：箭头函数不能显式地命名，尽管现代运行环境会将箭头函数所赋予的变量名作为函数名；箭头函数不能用作构造函数，也没有 prototype 属性，这意味着不能对它们使用 new 关键字；此外，箭头函数会绑定到所在词法作用域中，因此它们不会改变 this 的指向。

接下来我们将深入了解它们在语义上与传统函数的区别、声明箭头函数的多种方法以及实际用例。

2.2.1 词法作用域

由于箭头函数不会创建新的作用域，在箭头函数的函数体内，this、arguments 以及 super 均属于所在的父级作用域。思考以下示例，其中有一个 timer 计时器对象，对象上有一个 seconds 计数器属性和一个 start 方法，该方法就是用前面所讨论的方法定义语法创建的。然后我们开启这个计时器并等待几秒，接着会打印已经过去的秒数。

```
var timer = {
  seconds: 0,
  start() {
    setInterval(() => {
      this.seconds++
    }, 1000)
  }
}
timer.start()
setTimeout(function () {
  console.log(timer.seconds)
}, 3500)
// <- 3
```

如果不使用箭头函数，而是使用常规的匿名函数定义传入 setInterval 的函数，那么 this 将绑定到匿名函数的上下文，而不是 start 方法的上下文。此时，如果想要实现前面的 timer 计时器，需要在 start 方法的开头加上 var self = this 这样的声明语句，然后在 setInterval 函数内部引用 self。由此可见，使用箭头函数能够避免为保持上下文引用而额外增加的复杂性，只关注代码的功能即可。

同样，ES6 箭头函数的作用域绑定也意味着使用 .call、.apply、.bind 等方法调用函数时也无法改变 this 的指向。这一限制通常是很有用的，确保了上下文不会被修改。

再来看看以下示例。你认为 console.log 会输出什么呢？

```
function puzzle() {
  return function () {
    console.log(arguments)
  }
}
puzzle('a', 'b', 'c')(1, 2, 3)
```

答案是 1，2，3。因为 arguments 属于匿名函数的上下文，所以传入匿名函数的参数会被输出。

如果将上例中的匿名函数换为箭头函数，结果又会怎样呢？

```
function puzzle() {
  return () => console.log(arguments)
}
puzzle('a', 'b', 'c')(1, 2, 3)
```

在该示例中，arguments 对象属于 puzzle 函数的上下文，因为箭头函数并不会创建闭包。因此，输出结果为 'a'，'b'，'c'。

前面提过箭头函数有多种写法，但到目前为止，我们只使用了完整版的写法。其他几种写法是什么样的呢？

2.2.2 箭头函数的写法

我们来回顾一下目前所学的箭头函数语法。

```
var example = (parameters) => {
  // 函数体
}
```

如果箭头函数只有一个参数，那么可以省略圆括号。当然，这是可选的。将箭头函数传入其他方法时，这么做很有用，因为这样能够减少圆括号的数量，使得代码更简洁。

```
var double = value => {
  return value * 2
}
```

使用箭头函数声明简单函数时，完整写法较为烦琐，如前面的 double 方法。箭头函数的以下写法省略了函数体，使用 value * 2 表达式代替了整个函数体。当函数被调用时，表达式会进行计算，结果会作为返回值返回。此时 return 语句是隐式的，并且不需要使用花括号包裹函数体，只用一个表达式就行了。

```
var double = (value) => value * 2
```

注意，还可以同时使用隐式括号和隐式返回值，这样箭头函数会更加简洁。

```
var double = value => value * 2
```

隐式返回对象字面量

要想隐式返回对象字面量，需要使用圆括号将对象字面量包裹起来。否则，编译器会将花括号当作函数体的开始和结束标志。

```
var objectFactory = () => ({ modular: 'es6' })
```

在以下示例中，JavaScript 就将花括号当成了箭头函数的函数体。此外，number 会被当作一个 label[2]，value 表达式则没有任何作用。因为函数体没有返回任何内容，所以映射得到的值均为 undefined。

```
[1, 2, 3].map(value => { number: value })
// <- [undefined, undefined, undefined]
```

如果隐式返回的对象字面量具有多个属性，则编译器无法识别第二个属性，因此会抛出 SyntaxError 异常。

```
[1, 2, 3].map(value => { number: value, verified: true })
// <- SyntaxError
```

在对象字面量外添加圆括号即可解决该问题，这样编译器不会再将其当作函数体，此时的对象声明表达式即为我们想要隐式返回的对象字面量。

```
[1, 2, 3].map(value => ({ number: value, verified: true }))
/* <- [
  { number: 1, verified: true },
  { number: 2, verified: true },
  { number: 3, verified: true }]
*/
```

现在你应该已经理解什么是箭头函数，接下来我们再看看箭头函数的优点以及正确的用法。

2.2.3　优点和用例

一般来说，我们不应该盲目地使用 ES6 的特性。相反，最好在使用每个特性前仔细思考一下，使用新特性能否真的提高代码的可读性和可维护性。ES6 特性并不总是比现有特性好，最好不要随意使用它们。

箭头函数并不适用于某些情况。比如，当一个函数包含很多行代码时，使用箭头函数并不能对代码起到改进作用。箭头函数更适合简短实例，其中 function 关键字和语法模板占函数表达式的很大一部分。

为函数适当命名能使人更容易理解其含义。箭头函数不能显式地命名，但是可以通过赋给

注 2：label 用于定义指令。它可以用于 goto 语句，以指明需要跳转的指令；也可以用在 break 语句中表示所要跳出的序列；还可以用于 continue 语句，表示想要执行的序列。

其他变量隐式地命名。在以下示例中，我们将箭头函数赋给了 throwError 变量。当调用该函数的过程中发生错误时，调用栈能够正确地定位到 throwError。

```
var throwError = message => {
  throw new Error(message)
}
throwError('this is a warning')
<- Uncaught Error: this is a warning
  at throwError
```

当需要定义任何情况下词法作用域都不改变的匿名函数时，箭头函数很适合，并且在某些情况下，它还可以使代码更加整洁。在大多数函数式编程的情况下，箭头函数也特别有用，如使用数组对象的 .map、.filter 或者 .reduce 方法时，具体示例如下所示。

```
[1, 2, 3, 4]
  .map(value => value * 2)
  .filter(value => value > 2)
  .forEach(value => console.log(value))
// <- 4
// <- 6
// <- 8
```

2.3 解构

解构是 ES6 中最灵活且最有表现力的特性之一。同时，它也最为简单。它可以将对象的属性值绑定到任意数量的变量。解构可以用于对象、数组以及函数参数列表。我们从对象的解构开始逐一介绍。

2.3.1 对象的解构

假设我们有一个程序，其中有一些漫画角色，Bruce Wayne 是这些角色中的一个，我们想要引用描述 Bruce Wayne 的对象中的属性。以下是描述蝙蝠侠（Batman）的示例对象。

```
var character = {
  name: 'Bruce',
  pseudonym: 'Batman',
  metadata: {
    age: 34,
    gender: 'male'
  },
  batarang: ['gas pellet', 'bat-mobile control', 'bat-cuffs']
}
```

如果想要声明一个 pseudonym 变量并引用 character.pseudonym 的值，我们可能会写出如下的 ES5 代码。如果需要在多个地方引用 pseudonym，你肯定希望避免多次输入 character.pseudonym。

```
var pseudonym = character.pseudonym
```

如果在赋值中使用解构，上述示例的语法会变得更加清晰。正如你将看到的，使用解构赋值时，不需要重复输入两次 pseudonym 仍可以清晰表达含义。以下语句等价于前面的 ES5 代码。

```
var { pseudonym } = character
```

通过 var 声明时，使用逗号分隔能一次声明多个变量。同样，在解构表达式的花括号中也可以同时声明多个变量。

```
var { pseudonym, name } = character
```

同样，我们可以在同一个 var 语句中同时使用常规的变量声明和解构。这种用法一开始看起来可能会很奇怪，而且还要看所使用的 JavaScript 代码样式规范是否允许在单个语句中声明多个变量。无论如何，我们还是可以从这一点看出解构语法的灵活性。

```
var { pseudonym } = character, two = 2
```

如果想要提取 pseudonym 属性，并将其声明为 alias 变量，那么可以使用以下这种称为**别名**的解构语法。除了 alias，还可以使用任何其他合法的变量名。

```
var { pseudonym: alias } = character
console.log(alias)
// <- 'Batman'
```

别名语法看起来似乎并不比 ES5 写法简单（alias = character.pseudonym）。但解构支持深度结构，这就很有用了，如下所示。

```
var { metadata: { gender } } = character
```

类似以上这种情况，当需要解构一个嵌套较深的属性值时，使用别名能够更清晰地表达所有解构的属性名。试想一下，gender 并不能像 characterGender 一样清晰表达所指的内容。

```
var { metadata: { gender: characterGender } } = character
```

上述这种情况很常见，因为属性通常都是基于其宿主对象命名的。character.metadata.gender 的含义很清楚，而单独使用 gender 则可以表示很多内容，因此在解构赋值中使用 characterGender 这样的别名能够将上下文含义带入变量名。

在 ES5 中访问一个不存在的属性时会返回 undefined。

```
console.log(character.boots)
// <- undefined
console.log(character['boots'])
// <- undefined
```

解构中同样如此。当进行解构的属性不存在时，声明的解构变量也会得到 undefined。

```
var { boots } = character
console.log(boots)
// <- undefined
```

通常情况下，访问 `null` 或 `undefined` 的属性会报错。与此类似，当解构声明中嵌套属性的父对象是 `null` 或 `undefined` 时，也会抛出异常。

```
var { boots: { size } } = character
// <- Exception
var { missing } = null
// <- Exception
```

由于解构主要是语法糖，查看以下的 ES5 等价代码就能够清楚地知道为何上述代码会抛出异常。

```
var nothing = null
var missing = nothing.missing
// <- Exception
```

在解构中，我们可以为这些值为 `undefined` 的属性解构提供默认值。默认值可以是任何类型：数值、字符串、函数、对象，或者对其他变量的引用等。

```
var { boots = { size: 10 } } = character
console.log(boots)
// <- { size: 10 }
```

默认值也适用于嵌套属性解构。

```
var { metadata: { enemy = 'Satan' } } = character
console.log(enemy)
// <- 'Satan'
```

当与别名结合使用时，应该将别名放在前面，默认值放在后面，如下所示。

```
var { boots: footwear = { size: 10 } } = character
```

我们可以在解构模式中使用可计算属性名语法。不过，这种情况下必须提供一个别名作为变量名。因为可计算属性名中允许使用任意表达式，所以编译器无法推断出变量的名称。以下示例使用别名 `characterBoots` 和可计算属性从 `character` 对象中提取了 `boots` 属性。

```
var { ['boo' + 'ts']: characterBoots } = character
console.log(characterBoots)
// <- true
```

这一写法并没有太大用处，因为 `characterBoots = character[boots]` 比 `{ [boots]: characterBoots } = character` 更简单，也更符合表达习惯。也就是说，可计算属性名在声明对象字面量的属性名时很有用，但在解构赋值中恰恰相反。

以上就是对象解构的相关内容。数组的解构又是怎样的呢？

2.3.2 数组的解构

数组的解构语法和对象解构类似。以下示例展示了如何将 `coordinates` 数组解构成 `x` 和 `y` 两个变量。可以看到，这里使用了方括号，而不是花括号，这就表示我们在使用数组解构，而不是对象解构。解构允许我们在不显式引用索引的情况下清晰地为数组中的值命名，即不需要使用 `x = coordinates[0]` 这样的代码。

```
var coordinates = [12, -7]
var [x, y] = coordinates
console.log(x)
// <- 12
```

解构数组时可以跳过不感兴趣或不需要引用的值。

```
var names = ['James', 'L.', 'Howlett']
var [ firstName, , lastName ] = names
console.log(lastName)
// <- 'Howlett'
```

与对象解构类似，数组解构也可以设定默认值。

```
var names = ['James', 'L.']
var [ firstName = 'John', , lastName = 'Doe' ] = names
console.log(lastName)
// <- 'Doe'
```

在 ES5 中，当需要交换两个变量的值时，通常需要借助第三个临时变量。如下所示。

```
var left = 5
var right = 7
var aux = left
left = right
right = aux
```

解构可以帮助我们避免声明 `aux` 变量，专注于原本的意图。再次强调，解构能够使我们的表达更加清晰有效。

```
var left = 5
var right = 7
[left, right] = [right, left]
```

接下来我们将讨论有关解构的最后一项内容，即函数的参数。

2.3.3 函数参数的默认值

ES6 中的函数参数也能够指定默认值。以下示例中的 `exponent` 参数定义了一个最常用的默认值。

```
function powerOf(base, exponent = 2) {
  return Math.pow(base, exponent)
}
```

箭头函数的参数也可以指定默认值。当为箭头函数的参数指定默认值时，哪怕只有一个参数，也需要用圆括号将参数列表包裹起来。

```
var double = (input = 0) => input * 2
```

与某些编程语言不同，我们可以为任何一个参数设置默认值，而不是只能给最后一个参数设置。

```
function sumOf(a = 1, b = 2, c = 3) {
  return a + b + c
}
console.log(sumOf(undefined, undefined, 4))
// <- 1 + 2 + 4 = 7
```

在 JavaScript 中，向函数传递包含多个属性的 `options` 对象参数是再常见不过的。如果调用函数时没有传入，可以为其设定一个默认值对象 `options`，如下所示。

```
var defaultOptions = { brand: 'Volkswagen', make: 1999 }
function carFactory(options = defaultOptions) {
  console.log(options.brand)
  console.log(options.make)
}
carFactory()
// <- 'Volkswagen'
// <- 1999
```

该方法存在一个问题，如果 `carFactory` 的使用者传入一个 `options` 对象，则所有的默认值都会失效。

```
carFactory({ make: 2000 })
// <- undefined
// <- 2000
```

结合使用函数参数的默认值和解构能够解决这个问题。

2.3.4 函数参数的解构

与只提供一个默认值相比，更好的方法是解构整个 `options`，并在解构模式中为每个属性指定默认值。通过使用这个方法，我们不通过 `options` 对象就能引用 `options` 中的每个选项，但也因此不再能够直接引用 `options`，这在某些情况下可能会产生问题。

```
function carFactory({ brand = 'Volkswagen', make = 1999 }) {
  console.log(brand)
  console.log(make)
}
```

```
carFactory({ make: 2000 })
// <- 'Volkswagen'
// <- 2000
```

在这种情况下，如果使用者没有传入 options 对象，则默认值会再次缺失。也就是说，如果没有传入 options 对象参数，carFactory 就会报错。为 options 添加一个空对象作为默认值即可避免该问题，如下所示。这是因为解构空对象时已经提供了默认值。

```
function carFactory({
  brand = 'Volkswagen',
  make = 1999
} = {}) {
  console.log(brand)
  console.log(make)
}
carFactory()
// <- 'Volkswagen'
// <- 1999
```

除了默认值，我们还可以在函数参数中使用解构来描述函数能够处理的对象结构。思考以下代码，假设有一个包含多个属性的 car 对象。car 对象描述了其拥有者、类型、品牌、制造时间以及拥有者购买时的偏好。

```
var car = {
  owner: {
    id: 'e2c3503a4181968c',
    name: 'Donald Draper'
  },
  brand: 'Peugeot',
  make: 2015,
  model: '208',
  preferences: {
    airbags: true,
    airconditioning: false,
    color: 'red'
  }
}
```

如果只想在某个函数中提取对象的某些属性作为参数，可以通过解构提前显式地引用这些属性。这样做的好处是，看到函数声明时，就能知道函数需要使用哪些属性。

提前解构所需要的每个属性时，当输入不正确时，就很容易被发现。以下示例展示了如何在参数列表中指定需要的所有属性，这样我们对 getCarProductModel API 能够处理的参数就一目了然了。

```
var getCarProductModel = ({ brand, make, model }) => ({
  sku: brand + ':' + make + ':' + model,
  brand,
  make,
  model
```

```
})
getCarProductModel(car)
```

除了设置默认值和填充 options 对象，解构还有很多其他用处。接下来我们一起来看看。

2.3.5 解构的用例

当函数返回一个对象或数组时，解构能够更简洁地处理返回值。当函数返回一个包含一些坐标的对象，而我们只对其中的 x 和 y 感兴趣时，可以避免借助中间变量 point，这并不会影响代码的可读性，如下所示。

```
function getCoordinates() {
  return { x: 10, y: 22, z: -1, type: '3d' }
}
var { x, y } = getCoordinates()
```

默认值的使用能够避免重复。假设存在一个 random 函数，它会生成一个值在 min 和 max 之间的随机整数，默认生成 1~10 的值。与 Python 和 C# 等强类型语言中使用命名参数的实现方式相比，使用解构更有意思。这种模式能够为选项值定义默认值，并允许使用者分别覆盖，使用起来非常灵活。

```
function random({ min = 1, max = 10 } = {}) {
  return Math.floor(Math.random() * (max - min)) + min
}
console.log(random())
// <- 7
console.log(random({ max: 24 }))
// <- 18
```

解构也非常适用于正则表达式。解构使我们能够在不使用索引值的情况下命名匹配结果数组中的数据。以下示例通过正则表达式解析一个简单的日期，并使用解构将解析出来的值分别赋给对应的日期组成部分。匹配结果数组的第一个元素是原始输入，直接丢弃即可。

```
function splitDate(date) {
  var rdate = /(\d+).(\d+).(\d+)/
  return rdate.exec(date)
}
var [ , year, month, day] = splitDate('2015-11-06')
```

需要注意正则表达式未匹配的情况，此时匹配结果为 null。因此，最好在解构前进行错误处理，如下所示。

```
var matches = splitDate('2015-11-06')
if (matches === null) {
  return
}
var [, year, month, day] = matches
```

接下来我们关注一下扩展运算符和剩余参数。

2.4 剩余参数和扩展运算符

在 ES6 之前，处理任意数量的函数参数必须借助 `arguments`。`arguments` 不是一个数组，但具有 `length` 属性。通常来说，我们会使用 `Array#slice.call` 方法将 `arguments` 对象转换为真正的数组，如下所示。

```
function join() {
  var list = Array.prototype.slice.call(arguments)
  return list.join(', ')
}
join('first', 'second', 'third')
// <- 'first, second, third'
```

ES6 引入了剩余参数来更好地解决这一问题。

2.4.1 剩余参数

在函数的最后一个参数前添加 `...` 可以将该参数转变为一个特殊的“剩余参数”。当剩余参数是函数中的唯一参数时，它会获取所有传入函数的参数。这与上述使用 `.slice` 处理的结果相同，但不需要依赖于 `arguments`，直接在参数列表中指定即可。

```
function join(...list) {
  return list.join(', ')
}
join('first', 'second', 'third')
// <- 'first, second, third'
```

剩余参数前面的参数不会包含在 `list` 参数中。

```
function join(separator, ...list) {
  return list.join(separator)
}
join('; ', 'first', 'second', 'third')
// <- 'first; second; third'
```

注意，如果箭头函数中包含剩余参数，哪怕只有一个参数，也必须放置在圆括号内，否则会抛出 `SyntaxError` 异常。由下例可以看出，结合箭头函数和剩余参数能够写出更简洁的函数式表达式。

```
var sumAll = (...numbers) => numbers.reduce(
  (total, next) => total + next
)
console.log(sumAll(1, 2, 5))
// <- 8
```

可以看出，与以上代码相比，使用 ES5 实现相同的函数明显要更复杂。虽然以上实现方式较为简洁，但这样的 `sumAll` 函数会令没有使用过 `.reduce` 方法的使用者产生困扰，而且同

时使用两个箭头函数也会带来一定困扰。本书第二部分将探讨如何权衡这种情况。

```
function sumAll() {
  var numbers = Array.prototype.slice.call(arguments)
  return numbers.reduce(function (total, next) {
    return total + next
  })
}
console.log(sumAll(1, 2, 5))
// <- 8
```

接下来我们讨论扩展运算符。它也会用到 `...`，但目的不同。

2.4.2 扩展运算符

扩展运算符可以将可遍历对象转换为数组，能够在数组或函数调用中轻松展开表达式。以下示例使用 `...arguments` 将函数参数转换为一个数组字面量。

```
function cast() {
  return [...arguments]
}
cast('a', 'b', 'c')
// <- ['a', 'b', 'c']
```

扩展运算符可以将一个字符串分割成数组，数组中的元素为组成字符串的每个字符。

```
[...'show me']
// <- ['s', 'h', 'o', 'w', ' ', 'm', 'e']
```

还可以在扩展运算符的左右添加其他内容，结果和所期待的一样。

```
function cast() {
  return ['left', ...arguments, 'right']
}
cast('a', 'b', 'c')
// <- ['left', 'a', 'b', 'c', 'right']
```

扩展运算符非常适合用于拼接多个数组。以下示例展示了如何在数组字面量中展开任意数组，其中的元素会添加到相应位置。

```
var all = [1, ...[2, 3], 4, ...[5], 6, 7]
console.log(all)
// <- [1, 2, 3, 4, 5, 6, 7]
```

值得一提的是，扩展运算符不仅能用于数组和 `arguments`，还可以用于任何可遍历对象。可遍历是 ES6 新引入的一种机制，它允许我们将对象转换成可遍历的内容，第 4 章将深入探讨这一机制。

shift 操作和扩展运算

想要从一个数组的开头处获取一个或两个元素时，通常会使用 .shift 方法。虽然以下代码实现了这一功能，却不太容易理解，因为代码使用了两次 .shift 方法，且每次从数组开头获取的是不同的值。与 ES6 之前的情况很类似，侧重点在于如何让代码达到我们想要的目的。

```
var list = ['a', 'b', 'c', 'd', 'e']
var first = list.shift()
var second = list.shift()
console.log(first)
// <- 'a'
```

在 ES6 中，数组的解构和扩展运算符可以结合使用。以下代码和前面的代码很相似，但我们仅用一句代码即可实现，而且比重复使用 list.shift 方法表达得更清晰。

```
var [first, second, ...other] = ['a', 'b', 'c', 'd', 'e']
console.log(other)
// <- ['c', 'd', 'e']
```

通过使用扩展运算符，我们可以只关注需要实现的功能，而不用关心语言本身。ES6 的很多新特性都能够提高代码的表达力，并减少语言限制方面的时间花费。

在 ES6 之前，当某个函数调用的参数是动态的参数列表时，通常会使用 .apply 方法。这么做并不优雅，因为 .apply 要接受一个作为 this 的上下文参数，但我们又不想自己来传递它。

```
fn.apply(null, ['a', 'b', 'c'])
```

除了在数组中使用，扩展运算符还可以在函数调用中使用。以下示例展示了如何使用扩展运算符向 multiply 函数传递任意数值作为参数。

```
function multiply(left, right) {
  return left * right
}
var result = multiply(...[2, 3])
console.log(result)
// <- 6
```

与前面的数组字面量相同，在函数调用中扩展参数可以和常规参数一起使用。必要时可以使用任意数量的扩展参数。以下示例调用了 print 方法，并传入了一对常规参数和一对在参数列表中扩展开的数组。注意，使用剩余参数 list 能够获取所有传入的参数。扩展运算符和剩余参数可以在不增加代码的前提下更清晰地展示代码意图。

```
function print(...list) {
  console.log(list)
}
```

```
print(1, ...[2, 3], 4, ...[5])
// <- [1, 2, 3, 4, 5]
```

`.apply` 方法的一个不足之处是，使用 `new` 关键字实例化对象会非常冗长。以下示例用 `new` 和 `.apply` 创建了 `Date` 对象。大家都知道 JavaScript 中日期的月份是从 0 开始的，因此 11 是指 12 月。思考一下，为了实例化 `Date` 对象，我们的代码要扭曲成什么样。

```
new (Date.bind.apply(Date, [null, 2015, 11, 31]))
// <- Thu Dec 31 2015
```

如下所示，扩展运算符能够避免这些问题，我们只需要关注最重要的内容。在以下示例的 `new` 实例化过程中，`Date` 函数使用 `...` 扩展了动态的参数列表。

```
new Date(...[2015, 11, 31])
// <- Thu Dec 31 2015
```

下列表格总结了上述讨论的扩展运算符的用例。

用　　例	ES5	ES6
数组的连接	`[1, 2].concat(more)`	`[1, 2, ...more]`
将一个数组放入列表	`list.push.apply(list, items)`	`list.push(...items)`
解构	`a = list[0], other = list.slice(1)`	`[a, ...other] = list`
`new` 关键字和 `apply` 方法	`new (Date.bind.apply(Date,[null,2015,31,8]))`	`new Date(...[2015,31,8])`

2.5　模板字面量

模板字面量是对常规 JavaScript 字符串的巨大改进。例如，模板字面量不使用单引号或双引号进行声明，而是使用反引号“`”。

```
var text = `This is my first template literal`
```

由于模板字面量使用反引号作为定界符，在使用模板字面量声明字符串时，不需要再转义其中的 `'` 和 `"`，具体示例如下所示。

```
var text = `I'm "amazed" at these opportunities!`
```

最值得一提的是，模板字面量中可以插入 JavaScript 表达式。

2.5.1　字符串插值

模板字面量允许我们在模板中插入任意的 JavaScript 表达式。当执行到模板字面量表达式时，则会计算表达式并返回结果。以下示例在模板字面量内插入了一个 `name` 变量。

```
var name = 'Shannon'
var text = `Hello, ${ name }!`
```

```
console.log(text)
// <- 'Hello, Shannon!'
```

前面说过，除了变量，还可以使用任意的 JavaScript 表达式。我们可以将模板字面量中的表达式当作模板执行前定义的一个变量，执行结果是将各变量和其余字符串连接起来。这么做的好处是，代码更容易维护，不需要再手动拼接各字符串和表达式。此外，表达式中使用的变量、调用的函数等都必须在当前作用域内可用。

在模板字面量中插入多少表达式逻辑取决于自己的编码风格。以下代码实例化了一个 Date 对象，并将其格式化为对人友好的可读格式，然后插入模板字面量。

```
`The time and date is ${ new Date().toLocaleString() }.`
// <- 'the time and date is 8/26/2015, 3:15:20 PM'
```

同样，我们也可以在模板字面量中插入数学运算。

```
`The result of 2+3 equals ${ 2 + 3 }`
// <- 'The result of 2+3 equals 5'
```

甚至还可以嵌套模板字面量，因为它们也是合法的 JavaScript 表达式。

```
`This template literal ${ `is ${ 'nested' }` }!`
// <- 'This template literal is nested!'
```

模板字面量的另一个优势是支持多行字符串。

2.5.2　多行模板字面量

在使用模板字面量之前，如果想要在 JavaScript 中表示一个多行字符串，则必须借助转义符、字符串连接、数组，甚至是注释。以下代码总结了 ES6 前的多行字符串最常见的几种表示法。

```
var escaped =
'The first line\n\
A second line\n\
Then a third line'

var concatenated =
'The first line\n' `
'A second line\n' `
'Then a third line'

var joined = [
'The first line',
'A second line',
'Then a third line'
].join('\n')
```

ES6 中可以使用反引号来表示。模板字面量默认支持多行字符串。以下代码中没有 \n 转义、连接符，也不需要引入数组。

```
var multiline =
`The first line
A second line
Then a third line`
```

需要在一段 HTML 中插入多个变量时，多行字符串的优势就显现出来了。如果需要在模板中展示一个列表，那么可以直接遍历这个列表，将对应的内容添加到相应的标记，然后通过插值表达式返回 join 拼接后的结果。这使得在模板中声明子组件变得非常容易，如下所示。

```
var book = {
  title: 'Modular ES6',
  excerpt: 'Here goes some properly sanitized HTML',
  tags: ['es6', 'template-literals', 'es6-in-depth']
}
var html = `<article>
  <header>
    <h1>${ book.title }</h1>
  </header>
  <section>${ book.excerpt }</section>
  <footer>
    <ul>
      ${
        book.tags
          .map(tag => `<li>${ tag }</li>`)
          .join('\n      ')
      }
    </ul>
  </footer>
</article>`
```

上述代码会生成如下所示的 HTML 结构。我们可以看到，空格保留下来了[3]，并通过 join 方法中的一连串空格确保了 <li> 标签正确缩进。

```
<article>
  <header>
    <h1>Modular ES6</h1>
  </header>
  <section>Here goes some properly sanitized HTML</section>
  <footer>
    <ul>
      <li>es6</li>
      <li>template-literals</li>
      <li>es6-in-depth</li>
```

注 3：在使用多行模板字面量时，空格并不会自动保留。多数情况下，提供足够的缩进就能使其保留空格。因此，需要避免在代码块嵌套时产生不正确的缩进。

```
    </ul>
  </footer>
</article>
```

多行模板字面量在缩进方面存在问题。以下示例中的函数内包含一个模板字面量，模板字面量内的代码有着同样的缩进。我们期待的结果可能是没有缩进的，但最后字符串的前面会保留四个空格的缩进。

```
function getParagraph() {
  return `
    Dear Rod,

    This is a template literal string that's indented
    four spaces. However, you may have expected for it
    to be not indented at all.

    Nico
  `
}
```

我们可以通过以下的通用函数来移除结果字符串中的每一行的缩进，但这么做并不理想。

```
function unindent(text) {
  return text
    .split('\n')
    .map(line => line.slice(4))
    .join('\n')
    .trim()
}
```

有时，最好在插值表达式的结果插入模板前对其进行预处理。对于这些更高级的用例，需要用到另一种名为**标签模板**的模板字面量。

2.5.3 标签模板

通常情况下，JavaScript 中的 \ 具有特殊含义，代表转义。比如，`\n` 表示换行，`\u00f1` 表示 ñ。`String.raw` 标签模板可以使得转义字符不进行转义。以下的代码展示了如何使用 `String.raw`，其中 `\n` 并没有解释为换行。

```
var text = String.raw`"\n" is taken literally.
It'll be escaped instead of interpreted.`
console.log(text)
// "\n" is taken literally.
// It'll be escaped instead of interpreted.
```

模板字面量的前缀 `String.raw` 是一个标签模板，用于解析模板。标签模板接受一个数组参数和其他参数，数组中包含模板的每一个静态部分，其他参数对应每个表达式的计算结果。

思考以下代码中的标签模板。

```
tag`Hello, ${ name }. I am ${ emotion } to meet you!`
```

实际上，标签模板最终会解释为以下代码中的函数调用。

```
tag(
  ['Hello, ', '. I am ', ' to meet you!'],
  'Maurice',
  'thrilled'
)
```

依次取出模板中的每个部分并和相邻的表达式拼凑在一起，直到最终拼凑完模板中的所有部分，这就是结果字符串。如果不知道默认模板字面量 `tag` 内部是如何实现的，那么很难理解参数列表。因此，我们先探讨一下 `tag` 的内部实现。

以下代码是默认标签 `tag` 的一种可能实现方式。当不显式指定标签模板时，它的功能与模板字面量进行处理时相同。它会将 `parts` 数组归纳为一个值，以模板的第一部分开头，依次连接 `values` 中的值和模板中的内容，最终的值就是模板字面量的计算结果。我们用剩余参数语法获取 `...values`，从而更便捷地获取模板字面量中每个表达式的计算结果。由于表达式比较简单，我们还使用了隐式 `return` 的箭头函数。

```
function tag(parts, ...values) {
  return parts.reduce(
    (all, part, index) => all + values[index - 1] + part
  )
}
```

可以使用以下代码来尝试运行上述 `tag` 模板。我们可以看到，运行结果和不使用 `tag` 时相同，因为上述代码正是对默认模板行为的实现。

```
var name = 'Maurice'
var emotion = 'thrilled'
var text = tag`Hello, ${ name }. I am ${ emotion } to meet you!`
console.log(text)
// <- 'Hello Maurice, I am thrilled to meet you!'
```

标签模板还有很多其他用法。比如，可以将用户输入变为大写。以下的代码实现了这一功能。我们对 `tag` 进行了一些简单的修改，以便任意插入的字符串都会转为大写形式。

```
function upper(parts, ...values) {
  return parts.reduce((all, part, index) =>
    all + values[index - 1].toUpperCase() + part
  )
}
var name = 'Maurice'
var emotion = 'thrilled'
upper`Hello, ${ name }. I am ${ emotion } to meet you!`
// <- 'Hello MAURICE, I am THRILLED to meet you!'
```

标签模板还有一个更有用的用法，即可以用它来自动确保模板中插入表达式的安全性。假设有一个模板，其中所有表达式都是用户输入的内容，我们可以虚构一个 `sanitize` 库来移除 HTML 标签和类似的危害，从而阻止用户在网站中注入恶意的 HTML，防止跨域脚本（XSS，cross-site scripting）攻击。

```
function sanitized(parts, ...values) {
  return parts.reduce((all, part, index) =>
    all + sanitize(values[index - 1]) + part
  )
}
var comment = 'Evil comment<iframe src="http://evil.corp">
    </iframe>'
var html = sanitized`<div>${ comment }</div>`
console.log(html)
// <- '<div>Evil comment</div>'
```

看，邪恶的 `<iframe>` 差点就得逞了。介绍了这么多 ES6 的改进后，接下来我们探讨 ES6 中的 `let` 和 `const` 声明。

2.6 let和const声明

`let` 声明是 ES6 中最广为人知的特性之一。它和 `var` 声明功能相似，但有着不同的作用域规则。

关于作用域，JavaScript 有一套非常复杂的规则集，复杂到足以弄疯很多初次尝试弄懂 JavaScript 变量工作原理的程序员。最终，发现变量提升后，你才能逐渐理解变量的工作原理。**变量提升**指不管变量声明在代码的哪个位置，都会提升到所在作用域的顶部。参见以下示例。

```
function isItTwo(value) {
  if (value === 2) {
    var two = true
  }
  return two
}
isItTwo(2)
// <- true
isItTwo('two')
// <- undefined
```

虽然 `two` 是在 `if` 代码分支语句中声明的，但仍然可以在分支语句外访问到它，因此以上代码能够正常工作。这一行为是因为 `var` 声明会绑定到所在封闭作用域，可以是函数作用域或全局作用域。结合变量的提升，上述代码等同于以下代码。

```
function isItTwo(value) {
  var two
```

```
  if (value === 2) {
    two = true
  }
  return two
}
```

不管我们喜欢与否，与使用块级作用域的变量相比，变量提升很令人困惑。块级作用域通过花括号声明，而不是函数。

2.6.1 块级作用域和let声明

块级作用域允许我们在现有分支代码语句（如 if、for 或 while）的基础上嵌套任意新的 {} 块，以创建更深的作用域，而不需要声明新的函数。只要我们愿意，JavaScript 允许创建任意数量的块。

```
{{{{{ var deep = 'This is available from outer scope.'; }}}}}
console.log(deep)
// <- 'This is available from outer scope.'
```

使用 var 声明的变量是基于词法作用域的，仍然可以在 deep 变量声明所在块的外部访问到该变量，并且不会报错。但如果能够在以下几种情况下抛出异常，那么会更有用：

- 访问内部变量会破坏代码的封装性；
- 内部变量和外部变量没有任何关联；
- 同级的兄弟块中可能想要使用相同的变量名；
- 某个父级块中已经存在名称相同的变量，但仍然需要在内部使用该变量。

let 声明是 var 声明的一个替代方案。它遵循块级作用域规则，而不是默认的词法作用域规则。使用 var 时，只能通过嵌套函数来创建更深的作用域。但使用 let 时，新增一对花括号即可创建更深的作用域。这就意味着通过 {} 块即可创建新的作用域，无须创建新的函数。

```
let topmost = {}
{
  let inner = {}
  {
    let innermost = {}
  }
  // 在此处尝试访问innermost会抛出异常
}
// 在此处尝试访问inner会抛出异常
// 在此处尝试访问innermost会抛出异常
```

let 声明有一个非常有用的用法，如果在 for 循环中使用 let，则变量的作用域会封闭在循环体内，如下所示。

```
for (let i = 0; i < 2; i++) {
  console.log(i)
  // <- 0
  // <- 1
}
console.log(i)
// <- i is not defined
```

循环内声明的 `let` 变量会封闭在循环内的每一步，因此，在函数体内的异步函数调用中使用这些变量也能够如预期那样工作，这一点和使用 `var` 声明的变量恰恰相反。我们来看看具体的示例。

首先，我们来看一个关于 `var` 作用域工作原理的典型示例。以下示例中的 `i` 变量会绑定到 `printNumbers` 函数作用域，在循环添加每个超时回调时，它的值会一直增加到 `10`。等到每隔 100 毫秒执行每个回调函数时，`i` 的值是 `10`，因此每次都会输出 `10`。

```
function printNumbers() {
  for (var i = 0; i < 10; i++) {
    setTimeout(function () {
      console.log(i)
    }, i * 100)
  }
}
printNumbers()
```

相反，使用 `let` 声明会将变量绑定到块级作用域。虽然每次循环仍然会递增变量的值，但每次循环都会创建一个新的绑定。也就是说，每次添加超时回调时，每个回调函数都会持有一个保存当前变量 `i` 的绑定的引用，因此最终会输出期望结果：`0~9`。

```
function printNumbers() {
  for (let i = 0; i < 10; i++) {
    setTimeout(function () {
      console.log(i)
    }, i * 100)
  }
}
printNumbers()
```

除了以上内容，`let` 还涉及“暂时性死区”（TDZ，temporal dead zone）这一概念。

2.6.2　暂时性死区

毫无疑问，以下这样的代码段必然会抛出异常。从作用域开始到 `let` 声明的执行前，访问 `let` 声明的变量都会报错。这就是所谓的暂时性死区。

```
{
  console.log(name)
  // <- ReferenceError: name is not defined
```

```
  let name = 'Stephen Hawking'
}
```

如果在 `let name` 声明执行前访问变量 `name`，那么程序会抛出异常。在定义 `name` 变量前声明一个引用该变量的函数是没问题的，只要不在暂时性死区内执行该函数即可。在执行 `let name` 声明前，`name` 都处于暂时性死区。以下代码并不会报错，因为 `return name` 并没有在 `name` 的暂时性死区内执行。

```
function readName() {
  return name
}
let name = 'Stephen Hawking'
console.log(readName())
// <- 'Stephen Hawking'
```

相反，以下代码会报错，因为在 `name` 离开暂时性死区前就访问了 `name` 变量。

```
function readName() {
  return name
}
console.log(readName())
// ReferenceError: name is not defined
let name = 'Stephen Hawking'
```

注意，即使声明时没有对 `name` 赋值，上述示例的语义并不会发生改变。以下代码同样会抛出异常，因为在暂时性死区内访问了 `name` 变量。

```
function readName() {
  return name
}
console.log(readName())
// ReferenceError: name is not defined
let name
```

以下代码能够正常工作，因为它在离开暂时性死区后才访问 `name` 变量。

```
function readName() {
  return name
}
let name
console.log(readName())
// <- undefined
```

需要特别记住一点，在函数声明中访问暂时性死区中的变量是没问题的，只要访问处于暂时性死区中变量的语句在 `let` 声明语句之后执行即可。

暂时性死区的主要目的是更轻松地捕获错误，防止在用户代码声明变量前就访问变量，从而避免一些不可预期的行为。在 ES6 之前，由于变量的提升和不良的编码习惯，这一情况很常见。现在可以在 ES6 中很轻松地避免该问题。记住，提升仍然适用于 `let`，即变量

在作用域的开始就会创建，但会产生暂时性死区，这些变量在声明语句没有执行前无法访问，离开暂时性死区之后才能够访问。

探讨完暂时性死区后，现在是时候讨论 `const` 声明了。它与 `let` 相似，但也有很多不同。

2.6.3 const声明

与 `let` 相似，`const` 声明也遵循块级作用域，并存在暂时性死区。实际上，暂时性死区就是为了 `const` 实现的，之后为了保持统一，也应用于 `let` 了。`const` 之所以需要暂时性死区，是因为如果没有暂时性死区，则可以在 `const` 声明执行前给提升的 `const` 变量赋值，这样执行声明语句时就会报错。暂时性死区就是为了确保只在声明时对 `const` 进行赋值而实现的，这可以避免使用 `let` 的一些潜在问题，并且使其他依赖于暂时性死区的特性更容易实现。

以下示例表明，`const` 和 `let` 都遵循块级作用域规则。

```
const pi = 3.1415
{
  const pi = 6
  console.log(pi)
  // <- 6
}
console.log(pi)
// <- 3.1415
```

前面提过 `let` 和 `const` 有很多不同之处。第一个不同之处是使用 `const` 声明的变量必须在声明时就进行初始化，如下所示。

```
const pi = 3.1415
const e // 语法错误，缺少初始化
```

除了在声明时进行初始化，使用 `const` 声明的变量还无法重复赋值，即 `const` 完成初始化后无法再改变其值。在严格模式下，尝试改变 `const` 变量的值会报错。非严格模式下不会报错，但改变不生效，如下所示。

```
const people = ['Tesla', 'Musk']
people = []
console.log(people)
// <- ['Tesla', 'Musk']
```

需要注意的是，创建一个 `const` 变量并不意味着所赋的值不可改变。这是一个常见的混淆点，强烈建议你仔细阅读以下警告信息。

使用 const 声明的变量并不是不可改变的

使用 const 声明只是意味着所声明的变量会一直持有对同一个对象或基本值的引用，保持不变的只是这个引用。引用保持不变，但是引用指向的值并不是不可改变的。

以下示例表明，尽管 people 的引用不能改变，数组本身确实可以修改。如果数组是不可变的，那么不可能产生以下结果。

```
const people = ['Tesla', 'Musk']
people.push('Berners-Lee')
console.log(people)
// <- ['Tesla', 'Musk', 'Berners-Lee']
```

const 声明会禁止变量绑定到一个新的引用。以下代码从另一方面说明了该问题。我们用 const 创建了一个 people 变量，然后将这个变量赋给了一个普通的用 var 声明的 humans 变量。我们可以给 humans 重新赋值其他引用，因为它并不是使用 const 声明的。但我们不能将其他引用赋给 people，因为它是通过 const 声明的。

```
const people = ['Tesla', 'Musk']
var humans = people
humans = 'evil'
console.log(humans)
// <- 'evil'
```

如果想要确保值不变，可以使用 Object.freeze 这样的函数。使用 Object.freeze 可以禁止扩展传入的对象，如下所示。

```
const frozen = Object.freeze(
  ['Ice', 'Icicle', 'Ice cube']
)
frozen.push('Water')
// Uncaught TypeError: Can't add property 3
// object is not extensible
```

接下来我们一起讨论一下 const 和 let 的优势。

2.6.4 const和let的优势

永远不要为了使用新特性而使用新特性。ES6 的特性应该合理应用在能够真正提升代码可读性和可维护性的地方。很多情况下，let 声明能够简化一部分代码逻辑，比如需要在函数的顶部进行 var 声明，以确保变量的提升不会产生意想不到的结果时。使用 let 就可以不用在整个函数的顶部进行声明，而是在块级作用域的顶部声明，这可以减少思维从作用域顶部开始所要延续的范围。

使用 const 声明能够有效地避免一些问题。以下代码展示了一个常见的易错场景：将 items 的引用传递给 checklist 函数时，这个函数会返回一个 todo API，以用于操作传入的

items 引用。当 items 变量改为引用其他数组时，情况就完全不一样了，todo API 仍然会操作 items 之前引用的值，但 items 已经指向了其他值。

```
var items = ['a', 'b', 'c']
var todo = checklist(items)
todo.check()
console.log(items)
// <- ['b', 'c']
items = ['d', 'e']
todo.check()
console.log(items)
// <- 输出为['d', 'e']。如果items是常量的话，那么输出应该是['c']
function checklist(items) {
  return {
    check: () => items.shift()
  }
}
```

这种问题很难调试，可能需要花不少时间才能查出是引用改变导致出现问题。const 声明则可以防止这一错误发生，因为使用 const 会在运行时报错（严格模式下），从而能在产生问题时定位到问题所在。

使用 const 声明的另一个好处是能够显式地定义不可重复赋值的变量。const 表明变量引用的绑定是只读的，因此阅读代码时不用关心这些常量。

如果默认使用 const 来声明变量，使用 let 来声明需要重复赋值的变量，那么所有的变量都会遵循相同的作用域规则，这会使代码变得更易于理解。为什么提议默认使用 const 声明方式呢？因为它的功能明确且单一：禁止重新赋值、遵循块级作用域，并且不能在变量声明语句执行前访问变量。虽然 let 语句允许重新赋值，但它的行为和 const 类似，因此，如果需要一个可以重新赋值的变量，则可以选择使用 let 声明。

另一方面，var 声明方式更加复杂，由于需要遵循函数作用域规则，在分支语句中使用较为困难。var 允许重新赋值，且可以在变量声明语句执行前访问变量。因为 const 和 let 所做的事情更少、更简单，所以不推荐在现代 JavaScript 中使用 var，因而 var 声明在现代 JavaScript 代码库中并不常见。

本书默认使用 const，并在需要重新赋值时使用 let。第 9 章将探讨这样做的深层原因。

第 3 章

类、符号、对象和装饰器

前面探讨了 ES6 语法方面的基本改进，现在是时候探讨其他新增内容了，如类和符号。类基于传统的以类为基础的编程范式提供了一种表示原型链继承的新语法。与字符串、布尔值、数值一样，符号是一种新的 JavaScript 基本值类型。本章将探讨如何使用符号来定义协议。介绍完类和符号后，我们会简单探讨一下 ES6 中的 `Object` 新增的内置静态方法。

3.1　类

JavaScript 是一门基于原型的语言，类通常认为是原型继承的一种语法糖。原型继承与类之间的最主要区别是，类可以用 `extend` 继承其他类，这样我们就可以继承内置的 `Array` 对象，但在 ES6 之前这么做是很复杂的。

一些习惯其他范式的程序员并不熟悉原型链，有了 `class` 关键字之后，JavaScript 更容易为他们所接受。

3.1.1　使用类

要想学习新的语言特性，最好先找到语言中原有的实现，然后研究新的特性如何改进之前的用例。接下来我们先看一个简单的基于原型的 JavaScript 构造函数，然后将其与 ES6 中的类语法进行比较。

以下代码段定义了构造函数 `Fruit`，并在其原型上添加了两个方法。构造函数中有 3 个属性：表示名称的 `name`、表示水果热量的 `calories`，以及表示水果块数的 `pieces`（默认为 1）。

原型上有 .chop 和 .bite 两个方法。.chop 方法会将水果多切出一块。而传入 .bite 方法的 person 会吃掉一块水果，并且其 satiety 饱食度会增加，增加的值为剩余热量除以剩余的水果块数。

```
function Fruit(name, calories) {
  this.name = name
  this.calories = calories
  this.pieces = 1
}
Fruit.prototype.chop = function () {
  this.pieces++
}
Fruit.prototype.bite = function (person) {
  if (this.pieces < 1) {
    return
  }
  const calories = this.calories / this.pieces
  person.satiety += calories
  this.calories -= calories
  this.pieces--
}
```

以上代码比较简单，从中可以看出：有一个接受两个参数的构造函数、两个方法以及一些属性。以下代码创建了一个 Fruit 和一个 person，并且将水果切成 4 块，然后吃掉了其中 3 块。

```
const person = { satiety: 0 }
const apple = new Fruit('apple', 140)
apple.chop()
apple.chop()
apple.chop()
apple.bite(person)
apple.bite(person)
apple.bite(person)
console.log(person.satiety)
// <- 105
console.log(apple.pieces)
// <- 1
console.log(apple.calories)
// <- 35
```

以下代码使用了 class 语法，从中可以看出，constructor 函数显式地声明在 Fruit 类中，且方法声明使用了对象字面量的方法定义语法。对比 class 方法和前面基于原型的方法，能够发现：定义方法时，避免显式指向 Fruit.prototype 的引用可以减少重复代码。实际上，将整个声明写在 class 语句块内，还能够帮助阅读者理解整块代码的作用，更清晰地表达所声明类的意图。最后，将 constructor 显式声明为 Fruit 的成员方法使得 class 语法比基于原型的类语法更易理解。

```
class Fruit {
  constructor(name, calories) {
    this.name = name
    this.calories = calories
    this.pieces = 1
  }
  chop() {
    this.pieces++
  }
  bite(person) {
    if (this.pieces < 1) {
      return
    }
    const calories = this.calories / this.pieces
    person.satiety += calories
    this.calories -= calories
    this.pieces--
  }
}
```

可能你还没有注意到一个不容忽视的细节，即 `Fruit` 类中的方法声明之间没有用逗号分隔。其实，这并不是文字编辑的疏忽，`class` 语法就是这么设计的。这一差别能够避免我们误认为对象和类之间存在某些关联，但实际上它们并没有关联。此外，这一区别也会让类语法更适合做进一步改动，如添加公有和私有属性。

这段基于类的代码和前面所写的基于原型的代码等价。创建 `Fruit` 实例的方式一点也没有改变；`Fruit` 的 API 方法也没有任何改变。使用 `Fruit` 类也可以实例化一个苹果，将其切成更小的 4 块，然后吃掉 3 块。

值得注意的是，与函数声明不同，类声明不会提升到所在作用域的顶部。因此，在到达并执行类声明之前，我们无法实例化或进行其他存取操作。

```
new Person() // <- ReferenceError: Person is not defined
class Person {
}
```

除了前面所说的类声明语法，还可以通过表达式的方式声明类，这一点与函数声明和函数表达式相似。我们可以在表达式中省略类的名称，如下所示。

```
const Person = class {
  constructor(name) {
    this.name = name
  }
}
```

我们可以通过函数很轻松地返回类表达式，从而用最少的代价建立一个类的工厂函数。以下示例通过箭头函数动态创建了 `JakePerson` 类，箭头函数接受一个 `name` 参数，然后通过 `super()` 传入父类 `Person` 的构造函数中。

```
const createPersonClass = name => class extends Person {
  constructor() {
    super(name)
  }
}
const JakePerson = createPersonClass('Jake')
const jake = new JakePerson()
```

我们将继续深入探讨类的继承，现在先来更细致地研究一下属性和方法。

3.1.2 类的属性和方法

需要指出的是，class 声明中的 constructor 方法的声明是可选的。以下代码给出了一个完全有效的类声明和一个同名的空构造函数，它们等效。

```
class Fruit {
}
function Fruit() {
}
```

new Log() 的参数都会传入 constructor 方法，我们可以使用这些参数来初始化类的实例，如下所示。

```
class Log {
  constructor(...args) {
    console.log(args)
  }
}
new Log('a', 'b', 'c')
// <- ['a' 'b' 'c']
```

以下示例中的类展示了如何在每个实例的构造过程中创建并初始化一个名为 count 的属性。get next 方法表明 Counter 类的实例会有一个 next 属性，访问该属性时会返回该方法的调用结果。

```
class Counter {
  constructor(start) {
    this.count = start
  }
  get next() {
    return this.count++
  }
}
```

我们可以按照以下示例那样使用 Counter 类。每次访问 .next 属性时，count 的值会自动加 1。尽管很有用，但是这种用例通常更适合使用方法实现，而不是神奇的 get 属性存取器。而且我们要当心，不要滥用属性存取器，使用滥用了存取器的对象会令人感到很混乱。

```
const counter = new Counter(2)
console.log(counter.next)
// <- 2
console.log(counter.next)
// <- 3
console.log(counter.next)
// <- 4
```

当与 setter 方法一起使用时，存取器能够在对象和底层的数据存取之间搭建一条很有趣的桥梁。思考以下示例，我们定义了一个类，该类可以通过传入存储的 key 值从 localStorage 中存取 JSON 数据。

```
class LocalStorage {
  constructor(key) {
    this.key = key
  }
  get data() {
    return JSON.parse(localStorage.getItem(this.key))
  }
  set data(data) {
    localStorage.setItem(this.key, JSON.stringify(data))
  }
}
```

我们可以按照以下示例那样使用 LocalStorage 类。任何赋给 ls.data 的值都会转换为 JSON 对象字符串形式，然后存入 localStorage 中。读取属性时，可以通过同一个 key 值获取之前存储的内容，将其解析为 JSON 对象并返回。

```
const ls = new LocalStorage('groceries')
ls.data = ['apples', 'bananas', 'grapes']
console.log(ls.data)
// <- ['apples', 'bananas', 'grapes']
```

除了 getter 和 setter，我们还可以定义常规的实例方法，前面创建 Fruit 类时已经这么做了。以下示例创建了一个能吃 Fruit 的 Person 类，然后实例化了一个 fruit 和一个 person，并让 person 吃了 fruit。最终 person 的饱食度等于 40，因为他吃光了整个水果。

```
class Person {
  constructor() {
    this.satiety = 0
  }
  eat(fruit) {
    while (fruit.pieces > 0) {
      fruit.bite(this)
    }
  }
}
const plum = new Fruit('plum', 40)
const person = new Person()
person.eat(plum)
console.log(person.satiety)
// <- 40
```

有时需要在类自身中添加静态方法，而不是在实例上添加成员方法。在 ES6 之前，实例方法显式地添加在原型链上，而静态方法需要直接添加在构造函数上。

```
function Person() {
  this.hunger = 100
}
Person.prototype.eat = function () {
  this.hunger--
}
Person.isPerson = function (person) {
  return person instanceof Person
}
```

JavaScript 类允许通过 `static` 关键字定义 `Person.isPerson` 这样的静态方法。`static` 的使用方式与定义 `getter` 和 `setter` 时的 `get`、`set` 相似。

以下示例定义了 `MathHelper` 类，该类有一个 `sum` 方法，该方法通过 `Array#reduce` 计算所有传入的数值参数的总和。

```
class MathHelper {
  static sum(...numbers) {
    return numbers.reduce((a, b) => a + b)
  }
}
console.log(MathHelper.sum(1, 2, 3, 4, 5))
// <- 15
```

最后，值得一提的是，我们也可以定义静态属性存取器，如 `static get`、`static set`。在维护类的全局配置状态或在单例模式下使用类时，这一点尤为有用。当然，这种情况下，你可能更倾向于使用普通对象，而不是创建一个不会实例化或者只实例化一次的类。毕竟这就是 JavaScript，一个高度动态的语言。

3.1.3 类的继承

我们可以使用普通的 JavaScript 方法来实现 `Fruit` 类的继承。但阅读以下代码段后，你会发现：声明子类时，为了将参数传入父类以确保正确初始化子类，我们引入了较为难懂的知识点，如 `Parent.call(this)`，并将基于父类原型创建的新对象指定为子类的原型。网上可以找到很多有关原型继承的资料，这里不再详细讨论。

```
function Banana() {
  Fruit.call(this, 'banana', 105)
}
Banana.prototype = Object.create(Fruit.prototype)
Banana.prototype.slice = function () {
  this.pieces = 12
}
```

鉴于没有必要记住以上这种实现方式，而且 `Object.create` 是 ES5 中才存在的，以前的开发者习惯于借助类库来解决原型继承问题，如 Node.js 中的 `util.inherits` 方法，因为支持度更好，所以比 `Object.create` 更受喜爱。

```
const util = require('util')
function Banana() {
  Fruit.call(this, 'banana', 105)
}
util.inherits(Banana, Fruit)
Banana.prototype.slice = function () {
  this.pieces = 12
}
```

除了具有 `name` 属性和已经赋值的 `calories` 属性，以及额外的 `slice` 方法，`Banana` 构造函数的用法和 `Fruit` 没有什么不同。其中 `slice` 方法能够直接将香蕉切成 12 块。以下代码展示了吃了一块后的 `Banana` 的实际状态。

```
const person = { satiety: 0 }
const banana = new Banana()
banana.slice()
banana.bite(person)
console.log(person.satiety)
// <- 8.75
console.log(banana.pieces)
// <- 11
console.log(banana.calories)
// <- 96.25
```

类统一了原型继承，但到现在为止还存在很大的争议，因为很多库正在尝试用更简单的方式处理 JavaScript 中的原型继承。

`Fruit` 类已经可以被继承了。以下代码创建了继承自 `Fruit` 类的 `Banana` 类。此时，这种语法能够清晰地表达我们的意图，并且不需要彻底理解原型继承就能达到想要的结果。如果想要将参数转而传给底层的 `Fruit` 构造函数，可以使用 `super`。`super` 关键字还可以调用父类中的函数（如 `super.chop`），并不局限于父类的构造函数。

```
class Banana extends Fruit {
  constructor() {
    super('banana', 105)
  }
  slice() {
    this.pieces = 12
  }
}
```

虽然 `class` 关键字是静态的，但声明类时仍然可以利用 JavaScript 的灵活的函数式编程特性。返回构造函数的任何表达式都可以跟在 `extends` 之后。例如，我们可以创建一个构造函数工厂函数，然后将其作为基类。

以下代码定义了 createJuicyFruit 函数，通过 super 调用将水果的 name 和 calories 传到 Fruit 类中。然后我们要做的就是创建 Plum，使其继承中间类 JuicyFruit。

```
const createJuicyFruit = (...params) =>
  class JuicyFruit extends Fruit {
    constructor() {
      this.juice = 0
      super(...params)
    }
    squeeze() {
      if (this.calories <= 0) {
        return
      }
      this.calories -= 10
      this.juice += 3
    }
  }
class Plum extends createJuicyFruit('plum', 30) {
}
```

下一节将讨论符号。虽然符号不是一种迭代或流控制机制，但学习符号对理解迭代协议至关重要，后面的章节将详细讨论迭代协议。

3.2 符号

符号是 ES6 中新增的基本值类型，是 JavaScript 中的第 7 种数据类型。与字符串、数值一样，符号是一种独一无二的数据类型。与字符串、数值不同的是，符号不具备字面量表示形式，如字符串中的 'text'、数值中的 1。符号的主要目的是实现协议。例如，迭代协议使用符号来定义如何迭代对象，4.2 节将讨论这部分内容。

符号有 3 种类型，每种类型是用不同方式访问的：本地符号，通过内置符号包装对象创建，并通过存储引用或反射来访问；全局符号，通过另一种 API 创建，并跨代码域共享；“众所周知”的符号，内置于 JavaScript 中，用于定义内部语言行为。

我们将依次探讨这 3 种类型，并在探讨过程中探索可能的使用方式。我们先从本地符号开始。

3.2.1 本地符号

可以使用符号包装对象来创建符号。以下代码创建了 first 符号值。

```
const first = Symbol()
```

可以用 new 关键字来调用 Number 和 String，但 new 和符号一起使用时会抛出 TypeError 异常。这样能够避免类似 new Number(3) !== Number(3) 的错误和混淆行为。以下代码展示了错误信息。

```
const oops = new Symbol()
// <- TypeError, Symbol is not a constructor
```

为了便于调试，我们可以在创建符号时添加一些描述或说明。

```
const mystery = Symbol('my symbol')
```

与数值或字符串一样，符号值是不可变的。然而，与另两种值类型不同，符号值是独一无二的。描述并不会影响符号值的唯一性，如下所示。使用相同描述创建的符号值仍然是唯一的，互不相等。

```
console.log(Number(3) === Number(3))
// <- true
console.log(Symbol() === Symbol())
// <- false
console.log(Symbol('my symbol') === Symbol('my symbol'))
// <- false
```

符号的 `typeof` 结果为 `symbol`，这是 ES6 中新增的，如下所示。

```
console.log(typeof Symbol())
// <- 'symbol'
console.log(typeof Symbol('my symbol'))
// <- 'symbol'
```

符号可以用作对象的属性名。我们可以使用可计算属性直接将名为 `weapon` 的符号值添加到 `character` 对象中，如下所示。此外，为了访问对应的符号属性，我们需要保存创建属性的符号值的引用。

```
const weapon = Symbol('weapon')
const character = {
  name: 'Penguin',
  [weapon]: 'umbrella'
}
console.log(character[weapon])
// <- 'umbrella'
```

记住，从对象中读取属性名的传统方法无法获取符号类型的属性名。以下代码表明，`for..in`、`Object.keys` 和 `Object.getOwnPropertyNames` 均无法获取符号属性。

```
for (let key in character) {
  console.log(key)
  // <- 'name'
}
console.log(Object.keys(character))
// <- ['name']
console.log(Object.getOwnPropertyNames(character))
// <- ['name']
```

这一表现意味着 ES6 前的代码不会因为符号而出问题。同样，符号属性也不会出现在对象

JSON 字符串化的结果中，如下所示。

```
console.log(JSON.stringify(character))
// <- '{"name":"Penguin"}'
```

虽说如此，但符号并不是一种用于隐藏属性的安全机制。虽然使用反射和序列化方法时不会出现符号属性，但可以通过专有的方法获取到符号，如下所示。换句话说，符号并非不可枚举，只是简单地隐藏了。我们可以通过 `Object.getOwnPropertySymbols` 获取给定对象中所有用作属性名的符号值。

```
console.log(Object.getOwnPropertySymbols(character))
// <- [Symbol(weapon)]
```

现在我们已经知道符号的工作原理了，那么应该如何使用它呢？

3.2.2　符号的实际用法

JavaScript 库可以用符号将对象映射到 DOM 元素，例如，将日历的 API 对象关联到 DOM 元素。在 ES6 问世前，并没有很好的方式可以将 DOM 元素映射到对象。我们可以在 DOM 元素上添加一个指向 API 的属性，但这么做会污染 DOM 元素，因此并不是一种良好方式。同时，要小心使用其他库不会使用的属性，甚至语言本身将来也可能不会使用的属性。这样做还需要建立一个包含所有 DOM/API 条目的数组查找表。在长时间运行的应用中，数组查找表会变得越来越大，查询的速度也会越来越慢。

使用符号作为属性就不会存在这个问题。不用担心与将来的语言特性发生冲突，因为符号值是唯一的。以下代码使用了符号将 DOM 元素映射到日历 API 对象中。

```
const cache = Symbol('calendar')
function createCalendar(el) {
  if (cache in el) { // 检测Symbol值cache是否存在于el元素中
    return el[cache] // 直接使用cache，避免重新实例化
  }
  const api = el[cache] = {
    // 在这里编写日历API
  }
  return api
}
```

ES6 中内置的 `WeakMap` 可以将对象映射到其他对象，且不需要借助数组或在所有对应对象上添加额外的属性。与数组查找表不同，`WeakMap` 查询的时间复杂度是常量 $O(1)$。第 5 章将介绍 `WeakMap` 以及其他一系列 ES6 内置对象。

用符号定义协议

前面说过符号可以用来定义协议。协议是一种定义行为的通信契约或约定。更具体来说，如果一个库使用一个符号值，那么遵循这个库约定的对象也能够使用这个符号值。

思考以下代码，我们使用 toJSON 方法来决定对象通过 JSON.stringify 序列化的结果。从结果可以看出，character 对象字符串化的结果是 toJSON 方法返回的对象序列化后的结果。

```
const character = {
  name: 'Thor',
  toJSON: () => ({
    key: 'value'
  })
}
console.log(JSON.stringify(character))
// <- '"{"key":"value"}"'
```

相反，如果 toJSON 不是函数，那么原始的 character 对象会被序列化，包括 toJSON 属性，如下所示。之所以存在这种不一致性，是因为依赖常规属性来定义行为。

```
const character = {
  name: 'Thor',
  toJSON: true
}
console.log(JSON.stringify(character))
// <- '"{"name":"Thor","toJSON":true}"'
```

使用符号来实现 toJSON 会更好，因为这样就不会与其他对象属性名产生冲突。因为符号是唯一的、不会被序列化，且除了显式调用 Object.getOwnPropertySymbols 外不会暴露，所以符号很适合对象用来定义自己的序列化逻辑，然后通过 JSON.stringify 输出。以下代码用符号来定义 stringify 函数的序列化行为，我们可以将其当作一种替代方案。

```
const json = Symbol('alternative to toJSON')
const character = {
  name: 'Thor',
  [json]: () => ({
    key: 'value'
  })
}
stringify(character)
function stringify(target) {
  if (json in target) {
    return JSON.stringify(target[json]())
  }
  return JSON.stringify(target)
}
```

要想用符号值定义 json 行为，需要在对象字面量中使用可计算属性名。这样还可以确保该行为不会与用户定义的其他属性发生冲突，将来也不会与无法预见的语言特性发生冲突。此外，必须能够在 stringify 函数中访问符号值 json，这样才能自定义行为。我们可以通过一行代码就将符号值 json 暴露给 stringify 函数，如下所示。这也将 stringify 与改变其行为的符号值绑定在一起了。

```
stringify.as = json
```

暴露 stringify 函数也就暴露了符号值 stringify.as。这样用户就可以使用自定义符号简单修改对象，从而调整 stringify 的行为。

与向 stringify 函数传入可选参数截然不同，使用符号描述行为没有以下问题。首先，向函数添加可选参数会影响其公共 API，而修改函数的内部实现以支持符号并不会影响公共 API。使用 options 对象传入不同的属性会好一些，但每调用一次函数就传入一个 options 对象并不方便。

使用符号定义行为还有以下好处：只要修改符号属性值或利用了该行为的代码，就能增强或修改对象的行为。此外，使用符号作为属性无须担心与语言的新特性发生冲突。

除了本地符号，还有一些全局注册的符号，整个代码域内都能访问这些值。我们来一起看看具体有哪些。

3.2.3 全局符号注册表

代码域是指任何一种 JavaScript 执行上下文，如应用所在页面、页面中的 <iframe>、通过 eval 执行的脚本，以及各类 worker[1]（如 Web worker、service worker 和 shared worker）。这些执行上下文都有自己独有的全局对象。例如，定义在页面的 window 对象上的全局变量不可在 ServiceWorker 中使用。不过，全局符号注册表在所有代码域中是共享的。

以下两种方法可以访问运行环境下的全局符号注册表：Symbol.for 和 Symbol.keyFor。我们来看看它们的用途。

1. Symbol.for(key)

Symbol.for(key) 可以用来查找运行环境下的全局符号注册表中的 key 值。如果全局注册表中存在所传入 key 对应的符号值，则返回该值。如果不存在，则用传入的 key 新建一个并在全局注册。也就是说，Symbol.for(key) 是幂等的：使用给定的 key 查找符号值，如果没有，则创建一个并返回。

在以下代码段中，对 Symbol.for 的第一个调用创建了一个标识为 'example' 的符号值，并将其加入了全局注册表中，然后返回。第二个调用返回了同一个符号值，因为该 key 值已经在注册表中，返回值等于前一个符号值。

```
const example = Symbol.for('example')
console.log(example === Symbol.for('example'))
// <- true
```

全局注册表通过 key 保存符号。注意，当符号被创建并添加到全局注册表时，key 会用作

注 1：worker 是在浏览器中执行后台任务的一种方式。即使在不同的执行上下文中，启动程序也可以通过消息与其 worker 通信。

它的描述。试想一下，这些符号值是全局可用的，如果想要减少潜在的命名冲突，就需要给符号的 key 添加前缀，以标识库和组件。

2. Symbol.keyFor(symbol)

Symbol.keyFor(symbol) 能够返回一个符号类型的符号值在添加到全局注册表时所关联的 key。以下示例展示了这一用法。

```
const example = Symbol.for('example')
console.log(Symbol.keyFor(example))
// <- 'example'
```

注意，如果给定的符号值不在全局注册表中，那么该方法会返回 undefined。

```
console.log(Symbol.keyFor(Symbol()))
// <- undefined
```

还需要记住一点，即使描述相同，使用本地符号也不可能匹配全局注册中的符号值。这是因为本地符号值并不是全局注册的一部分，如下所示。

```
const example = Symbol.for('example')
console.log(Symbol.keyFor(Symbol('example')))
// <- undefined
```

到目前为止，我们已经学习了有关全局注册表的 API，接下来我们再看一些注意事项。

3. 最佳实践与注意事项

全局注册表意味着整个代码域内都可以访问符号值。全局注册表在任何代码域内返回的都是同一个对象的引用。在以下示例中，Symbol.for API 在页面和 <iframe> 中返回的是同一个符号值。

```
const d = document
const frame = d.body.appendChild(d.createElement('iframe'))
const framed = frame.contentWindow
const s1 = window.Symbol.for('example')
const s2 = framed.Symbol.for('example')
console.log(s1 === s2)
// <- true
```

使用全局可用的符号值时需要做好权衡。一方面，全局符号使得类库能够方便地暴露其中的符号值，另一方面，类库也可能会使用本地符号在其 API 上暴露符号值。显然，当符号需要在任意两个代码作用域（如 ServiceWorker 和 Web 页面）共享时，全局符号注册表就非常有用了。同时，使用全局符号注册表的 API 可以不必存储符号值的引用，因为同一个给定的 key 值会返回相同的符号值。记住，虽然这些符号值在整个运行环境均可用，但使用 each 或 contains 这样的通用符号值可能会产生意想不到的后果。

此外，还有另一种符号值：众所周知的内置符号。

3.2.4 众所周知的符号

前面我们已经讨论了如何用 Symbol 函数和 Symbol.for 创建符号值。接下来我们将讨论第三种，也是最后一种符号：众所周知的符号。这些符号值是语言自带的，而不是开发者自己创建的，它们提供了内部语言行为的钩子，允许我们扩展或定制 ES6 问世前无法操作的某些内部逻辑。

众所周知的符号能够在不破坏现有代码的情况下扩展语言，Symbol.toPrimitive 就是一个很好的示例。可以将一个函数赋给它，该函数将决定对象如何转换成基本值。函数接受一个可以为 'string'、'number' 或者 'default' 的 hint 参数，以指定所要转换成的初始类型。

```
const morphling = {
  [Symbol.toPrimitive](hint) {
    if (hint === 'number') {
      return Infinity
    }
    if (hint === 'string') {
      return 'a lot'
    }
    return '[object Morphling]'
  }
}
console.log(+morphling)
// <- Infinity
console.log(`That is ${ morphling }!`)
// <- 'That is a lot!'
console.log(morphling + ' is powerful')
// <- '[object Morphling] is powerful'
```

另一个示例是 Symbol.match。如果将一个正则表达式的 Symbol.match 属性设为 false，当传入 .startsWith、.endsWith 或者 .includes 时，该正则表达式会被当作字符串字面量。这三个方法是 ES6 中新增的字符串方法。.startsWith 用于判断字符串是否以另一个字符串开头。.endsWith 表明字符串是否以另一个字符串结尾。.includes 方法用于判断字符串是否包含另一个字符串，如果是，则返回 true。以下代码段通过 Symbol.match 将正则表达式变成了字符串形式，并将其与字符串进行比较。

```
const text = '/an example string/'
const regex = /an example string/
regex[Symbol.match] = false
console.log(text.startsWith(regex))
// <- true
```

如果没有通过 Symbol.match 修改正则表达式，那么 .startsWith 会抛出异常，因为该方法期望接受一个字符串参数，而不是正则表达式。

跨代码域共享但不在全局注册表中

众所周知的符号跨代码域共享。以下代码表明，当前窗口中的 Symbol.iterator 和 <iframe> 窗口中的引用相同。

```
const frame = document.createElement('iframe')
document.body.appendChild(frame)
Symbol.iterator === frame.contentWindow.Symbol.iterator
// <- true
```

需要注意的是，虽然众所周知的符号可以跨代码域共享，但它们并不在全局注册表中。以下代码表明，使用 `Symbol.iterator` 作为 key 在注册表中查询时，得到的值为 `undefined`。这表明该符号值并不在全局注册表中。

```
console.log(Symbol.keyFor(Symbol.iterator))
// <- undefined
```

`Symbol.iterator` 是最有用的众所周知的符号之一。在任意对象上使用该符号定义一个函数可以在不同的语言结构上迭代序列。下一章将深入讨论 `Symbol.iterator`，并将其与迭代器和迭代协议一起广泛使用。

3.3 对象的内置改进

第 2 章中介绍了对象字面量的语法方面的改进，并没有提及一些内置的 `Object` 静态方法。现在我们来看看这些方法。

除了前面提到的 `Object.getOwnPropertySymbols`，还有 `Object.assign`、`Object.is` 和 `Object.setPrototypeOf`。

3.3.1 使用`Object.assign`扩展对象

为配置对象设置默认值是再常见不过的事情。通常来说，类库和设计良好的组件接口都有合理的缺省值，以满足最频繁的使用场景。

例如，一个 Markdown 类库只要提供一个 `input` 参数就可以将 Markdown 转为 HTML。只进行 Markdown 解析是最常用的使用场景，因此类库不要求使用者提供其他任何选项。但是类库可能会支持一些不同的选项，以调整解析行为。这些选项可能是允许 `<script>` 或 `<iframe>` 标签的选项、在代码块中用 CSS 高亮关键字的选项。

试想一下，我们需要提供以下这些默认值。

```
const defaults = {
  scripts: false,
  iframes: false,
  highlightSyntax: true
}
```

其中一种方式是通过解构将 `defaults` 对象设为 `options` 参数的默认值。这种情况下，当想要设置选项时，用户必须为每个选项提供值。

```
function md(input, options=defaults) {
}
```

因此，默认值必须以某种方式与用户提供的配置合并。Object.assign 能够进行这一合并操作，如下所示。第一个参数为空对象 {}，第二个参数 default 中的值会复制到空对象中，然后再复制 options，最终返回改变后的空对象。这样 config 对象中就会有所有的默认值以及用户提供的配置。

```
function md(input, options) {
  const config = Object.assign({}, defaults, options)
}
```

理解 Object.assign 的目标

Object.assign 函数会改变它的第一个参数。参数格式是 (target, ...sources)。每个源对象都会应用到目标对象上，源对象依次应用，源对象中的属性依次应用。

思考以下场景，相较于在 Object.assign 中传递一个空对象作为第一个参数，我们直接传入了 defaults 和 options。这样会改变 defaults 对象的内容，由于 Object.assign 会改变第一个对象，这一过程会导致 defaults 中的部分默认值丢失，并额外增加一些错误的默认值。首次调用函数 md 得到的结果与上例中的结果相同，但它在该过程中改变了 defaults，从而影响了后续的调用。

```
function md(input, options) {
  const config = Object.assign(defaults, options)
}
```

因此，最好每次调用 Object.assign 时都传入一个全新的对象作为第一个参数。

对于有默认值的任意属性，如果用户提供了新的值，则以用户提供的值为准。Object.assign 的工作原理如下所述。首先，Object.assign 获取传入的第一个参数，我们可以称其为 target；然后获取其他参数，我们称其为 sources。对于 sources 中的每一个源对象，其中所有的属性值都会被遍历并赋给 target 对象。最终，最后一个源对象（以下示例中为 options 对象）会重写前面的赋值，具体代码如下所示。

```
const defaults = {
  first: 'first',
  second: 'second'
}
function applyDefaults(options) {
  return Object.assign({}, defaults, options)
}
applyDefaults()
// <- { first: 'first', second: 'second' }
applyDefaults({ third: 3 })
// <- { first: 'first', second: 'second', third: 3 }
applyDefaults({ second: false })
// <- { first: 'first', second: false }
```

该语言中还没有 Object.assign 时，这个功能在用户间有很多相似的实现方式，具体名称有 assign 或 extend 等。Object.assign 方法统一了这些可选方案。

需要注意的是，Object.assign 只会遍历自身可数的属性，其中包括字符串和符号属性。

```
const defaults = {
  [Symbol('currency')]: 'USD'
}
const options = {
  price: '0.99'
}
Object.defineProperty(options, 'name', {
  value: 'Espresso Shot',
  enumerable: false
})
console.log(Object.assign({}, defaults, options))
// <- { [Symbol('currency')]: 'USD', price: '0.99' }
```

然而，Object.assign 并不能照顾到每一种需求。很多用户间的实现方法能够支持深度赋值，但 Object.assign 并不会递归对象。值为对象的属性会直接赋给 target，而不会被递归赋值。

在以下示例中，你可以期望 f 属性被添加到 target.a 上，同时 b.c 和 b.d 原封不动，但使用 Object.assign 时，b.c 和 b.d 则会丢失。

```
Object.assign({}, { a: { b: 'c', d: 'e' } }, { a: { f: 'g' } })
// <- { a: { f: 'g' } }
```

同样，数组也不会被特别对待。如果你期待 Object.assign 能够进行递归，那么以下示例的结果就会出乎你的意料，你所期待的 'd' 并不会出现在结果数组的第三个位置上。

```
Object.assign({}, { a: ['b', 'c', 'd'] }, { a: ['e', 'f'] })
// <- { a: ['e', 'f'] }
```

撰写本书时有一个处于阶段 3 的 ECMAScript 提案[2]，该提案实现了在对象中进行扩展，这与在数组中扩展可遍历对象相似。在一个对象中扩展另一个对象与 Object.assign 函数的调用效果等价。

以下代码展示了在对象中扩展另一个对象的用法，并给出了 Object.assign 方式作为对比。正如你看到的，使用对象扩展更加简洁，情况允许时，应该优先选择该方式。

```
const grocery = { ...details }
// Object.assign({}, details)
const grocery = { type: 'fruit', ...details }
// Object.assign({ type: 'fruit' }, details)
const grocery = { type: 'fruit', ...details, ...fruit }
```

注 2：可以在 GitHub（https://github.com/tc39/proposal-promise-finally）中查看该草案。

```
// Object.assign({ type: 'fruit' }, details, fruit)
const grocery = { type: 'fruit', ...details, color: 'red' }
// Object.assign({ type: 'fruit' }, details, { color: 'red' })
```

与对象扩展对应，该提案还包含对象剩余属性，这与数组中的剩余参数模式相似。我们可以在解构对象时使用对象剩余属性。

以下示例表明，我们可以使用对象剩余属性得到一个对象，该对象只包含未在参数列表中显式指明的属性。注意，与数组剩余参数模式相同，对象剩余属性只能位于解构的最后位置。

```
const getUnknownProperties = ({ name, type, ...unknown }) =>
  unknown
getUnknownProperties({
  name: 'Carrot',
  type: 'vegetable',
  color: 'orange'
})
// <- { color: 'orange' }
```

在变量声明语句中解构对象时，我们也可以使用相似的方式。在以下示例中，没有显式解构的每个属性都会被放到 `meta` 对象中。

```
const { name, type, ...meta } = {
  name: 'Carrot',
  type: 'vegetable',
  color: 'orange'
}
// <- name = 'Carrot'
// <- type = 'vegetable'
// <- meta = { color: 'orange' }
```

第 9 章将深入讨论对象的剩余属性与扩展。

3.3.2 使用`Object.is`进行对象比较

`Object.is` 方法与严格相等比较运算符 `===` 有些不同。大部分情况下，`Object.is(a, b)` 与 `a === b` 等价。但是有两种情况不同：`NaN`；`-0` 和 `+0`。该算法在 ECMAScript 规范中被称为 `SameValue`。

当 `NaN` 和 `NaN` 比较时，严格相等运算符会返回 `false`，因为 `NaN` 不等于它自身。但是 `Object.is` 方法则会返回 `true`。

```
NaN === NaN
// <- false
Object.is(NaN, NaN)
// <- true
```

类似地，当 -0 与 +0 比较时，=== 运算符得到的结果是 true，但 Object.is 返回 false。

```
-0 === +0
// <- true
Object.is(-0, +0)
// <- false
```

这些差别看起来不大，但由于 NaN 有一些独特的行为，如 typeof NaN 的结果是 'number'，处理 NaN 时总是很麻烦，而且 NaN 不等于其自身。

3.3.3 Object.setPrototypeOf

顾名思义，Object.setPrototypeOf 的功能是设置一个对象的原型引用为另一个对象。相较于使用遗留特性 _proto_ 来设置原型，更推荐使用 Object.setPrototypeOf。

在 ES6 问世前，ES5 引入了 Object.create。我们可以使用该方法创建一个新的对象，新对象以传入的原型参数为原型，如下所示。

```
const baseCat = { type: 'cat', legs: 4 }
const cat = Object.create(baseCat)
cat.name = 'Milanesita'
```

但是 Object.create 只适用于创建新的对象。相反，我们可以使用 Object.setPrototypeOf 来改变已有对象的原型，如下所示。

```
const baseCat = { type: 'cat', legs: 4 }
const cat = Object.setPrototypeOf(
  { name: 'Milanesita' },
  baseCat
)
```

注意，相较于 Object.create，使用 Object.setPrototypeOf 可能会导致很多性能问题。因此，当决定在整个代码库中使用 Object.setPrototypeOf 时，需要认真考虑。

性能问题

使用 Object.setPrototypeOf 修改对象的原型是很耗费性能的操作。以下是 Mozilla 开发者网络文档中关于此事的说法。

> 鉴于现代 JavaScript 引擎优化属性访问的特性，在任何浏览器和 JavaScript 引擎中修改对象的原型都是一个很慢的操作。对于改变继承的性能的影响是很微妙与深远的，不仅仅是 obj.__proto = ... 语句的时间消耗，还会影响那些访问原型被改变的对象的所有代码。如果比较在意性能，那么你最好避免给对象设置原型。相反，可以使用 Object.create 创建一个基于期望原型的新对象。
>
> —— Mozilla 开发者网络

3.4 装饰器

装饰器并不是一个新的概念。在现代编程语言中，这种模式相当普遍：C# 中有**特性**、Java 中称其为**注解**，Python 中则称之为**装饰器**，等等。TC39 工作中有一个关于装饰器的提案[3]，现在处于阶段 2。

3.4.1 初识JavaScript装饰器

JavaScript 装饰器语法与 Python 相似。JavaScript 装饰器可以应用于类以及任何静态定义的属性，如对象字面量或类声明中的属性，哪怕它们是 `get`/`set` 存取器或 `static` 属性。

根据该提案，**装饰器**的语法是 `@` 后跟一系列点连接的标识符[4]以及一个可选参数列表。以下是一些示例。

- `@decorators.frozen` 是一个合法装饰器。
- `@decorators.frozen(true)` 是一个合法装饰器。
- `@decorators().frozen()` 存在语法错误。
- `@decorators['frozen']` 存在语法错误。

类声明和类成员可以添加 0 到多个装饰器。

```
@inanimate
class Car {}

@expensive
@speed('fast')
class Lamborghini extends Car {}

class View {
  @throttle(200) // 最多每200毫秒核对一次
  reconcile() {}
}
```

装饰器是通过函数实现的。成员装饰器函数接受成员描述符并返回成员描述符。成员描述符与属性描述符相似，但形式不同。以下即为装饰器提案中所定义的成员描述符接口。可选参数 `finisher` 函数接受类构造函数，允许我们执行与被修饰属性所在类相关的操作。

```
interface MemberDescriptor {
  kind: "Property"
  key: string,
  isStatic: boolean,
  descriptor: PropertyDescriptor,
  extras?: MemberDescriptor[]
```

注 3：可以在 GitHub（https://mjavascript.com/out/decorators）中查看该草案。

注 4：不允许通过 `[]` 访问属性，因为这样编译器会难以消除语法歧义。

```
  finisher?: (constructor): void;
}
```

以下示例定义了一个 readonly 成员装饰器函数，从而将被装饰的成员状态变为不可写。借助对象剩余属性与对象扩展操作，我们可以将属性描述符修改成不可写状态，并保持其余成员描述符不变。

```
function readonly({ descriptor, ...rest }) {
  return {
    ...rest,
    descriptor: {
      ...descriptor,
      writable: false
    }
  }
}
```

类装饰器函数接受 3 个参数：被装饰的类构造函数 ctor；如果被修饰的类继承了其他类，则包含父类的 heritage；带有被修饰类的成员描述符列表的 merbers 数组。

对被装饰类的每个成员描述符调用上述 readonly 成员装饰器可以实现类级的 readonlyMembers 装饰器，如下所示。

```
function readonlyMembers(ctor, heritage, members) {
  return members.map(member => readonly(member))
}
```

3.4.2 装饰器叠加及不变性提醒

为避免不变性问题，你可能会试图从装饰器中返回一个新的属性描述符，而不修改原始描述符。虽然出发点是好的，但可能会产生意想不到的效果，因为可能会对同一个类或类成员进行多次装饰。

如果代码中有装饰器未考虑自己接受的 descriptor 参数而直接返回一个全新的 descriptor，那么在这个新的描述符返回之前，这些装饰器会失去之前的所有修饰。

在编写装饰器时，我们应该认真对待接收到的 descriptor，做到创建并返回一个基于所给 descriptor 原始参数的 descriptor。

3.4.3 用例：C#中的特性

很久以前，我是通过开源的 C# 项目 RunUO 中编写的 Ultima Online[5] 服务器仿真器了解到 C# 的。RunUO 是我碰到过的最好的项目之一，而且它是使用 C# 实现的。

注 5：*Ultima Online* 是一款几十年前的基于终极宇宙的虚拟角色扮演类游戏。

他们将服务器软件拆成一个可执行文件和一堆 .cs 文件。runuo 执行文件会在运行时编译这些 .cs 脚本，并将它们动态混合到应用中。这样一来，只要能修改这些 .cs 文件的脚本就行了，用不着 VS（包括 `msbuild`）或其他东西。因此，RunUO 成为了新手的完美学习环境。

RunUO 高度依赖反射。RunUO 的开发者为了让玩家可以自己定制花了不少功夫。玩家不需要研究程序就能改变游戏中的一些细节，如龙的火焰能够造成多大的伤害以及它喷射火球的频率。良好的开发体验是 RunUO 设计哲学的重点。复制一些怪物文件，使其继承 `Dragon` 类，重写几个属性来改变它的色调、伤害输出等，就可以创建一个新的物种。

与能够便捷创建新的怪物或**非玩家角色**（游戏中的俚语是 NPC）一样，他们也依赖反射为游戏管理员提供功能。管理员可以在游戏内运行命令，点击一个物品或怪物来查看或改变属性，而无须离开游戏（见图 3-1）。

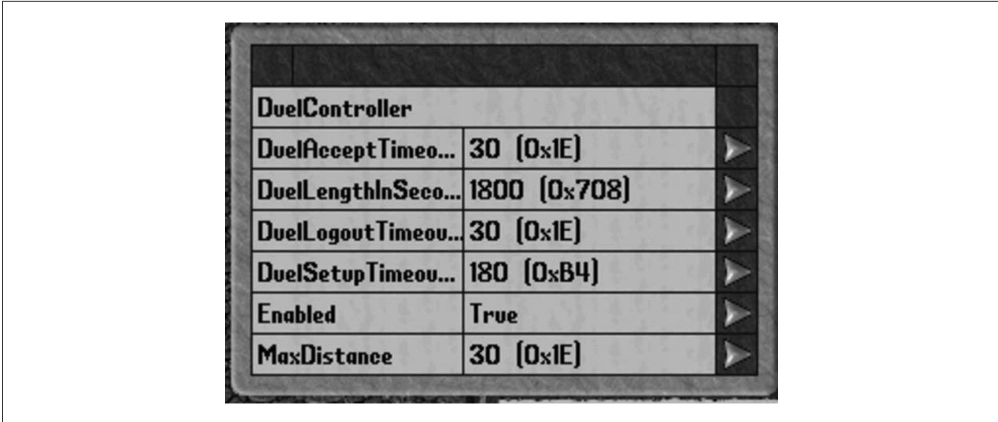


图 3-1：在 Ultima Online 客户端改变 RunUO 中的物品（角色）的属性

并不是类中的每个属性都可以在游戏内进行操作。有些属性仅限于内部使用，或者不支持运行时编辑。RunUO 有一个 `CommandPropertyAttribute` 装饰器 [6]，用于定义属性是否可以在游戏内编辑，同时要求指定读操作与写操作所需要的访问级别。这个装饰器在 RunUO 代码中几乎随处可见 [7]。

`PlayerMobile` 类用来控制玩家角色如何工作，它非常适合作为示例来说明这些特性。在游戏内，`PlayerMobile` 有很多属性对于管理员和审核人员都是可访问的 [8]。以下是一些 `getter` 和 `setter` 属性，但只有第一个应用了 `CommandProperty` 特性，因此该属性在游戏中对游戏管理者来说是可访问的。

注 6：可以在 RunUO 的 Git 仓库（https://mjavascript.com/out/runuo-attributes）中查看 `CommandPropertyAttribute` 的定义。

注 7：`CommandPropertyAttribute` 在整个代码库中广泛使用，仅在 RunUO 核心中就标记了 200 多个属性。

注 8：可以在 `PlayerMobile.cs` 类中找到许多 `CommandProperty` 属性的用法示例。

```
[CommandProperty(AccessLevel.GameMaster)]
public int Profession
{
  get{ return m_Profession }
  set{ m_Profession = value }
}

public int StepsTaken
{
  get{ return m_StepsTaken }
  set{ m_StepsTaken = value }
}
```

C# 特性和 JavaScript 装饰器的一个有趣区别是，C# 中的反射支持使用 `MemberInfo#getCustomAttributes` 从对象中提取所有自定义的属性。在显示允许管理员查看或修改游戏中物品属性的对话框时，RunUO 就使用该方法提取了游戏中可访问的每个属性。

3.4.4 在JavaScript中装饰属性

JavaScript 还没有从对象上获取自定义属性的方法，至少现有的草案中还没有。然而，JavaScript 是一种高度动态的语言，创建这种“标签”并不麻烦。用仅对管理员可见的属性装饰 `Dog` 与使用 C# 没什么不同。

```
class Dog {
  @commandProperty('game-master')
  name;
}
```

与 C# 相比，`commandProperty` 函数要稍微复杂一点。由于此时的 JavaScript 装饰器没有利用反射[9]，我们需要用一个运行时符号值来保存任意给定类的可控制属性数组。

```
function commandProperty(writeLevel, readLevel = writeLevel) {
  return ({ key, ...rest }) => ({
    key,
    ...rest,
    finisher(ctor) {
      const symbol = Symbol.for('commandProperties')
      const commandPropertyDescriptor = {
        key,
        readLevel,
        writeLevel
      }
      if (!ctor[symbol]) {
        ctor[symbol] = []
      }
      ctor[symbol].push(commandPropertyDescriptor)
```

注 9：现代 JavaScript 还没有考虑实现 JavaScript 装饰器的反射功能，因为这样需要引擎在内存中存储更多的元数据。但是我们可以使用符号和列表来代替使用原生反射。

```
    }
  })
}
```

只要有需要，Dog 类就可以有任意数量的可控制属性，并且这些属性都能通过一个符号取得。我们可以使用以下函数来查找给定类的可控制属性，该函数通过符号获取可控制的属性，并提供默认值 []，而且返回原始列表的副本，以防用户意外更改。

```
function getCommandProperties(ctor) {
  const symbol = Symbol.for('commandProperties')
  const properties = ctor[symbol] || []
  return [...properties]
}
getCommandProperties(Dog)
// <- [{ key: 'name', readLevel: 'game-master',
// writeLevel: 'game-master' }]
```

这样我们就可以遍历已知的安全可控制属性，并在运行时通过简单的 UI 呈现这些属性的编辑界面。装饰器最适合用来实现这种将某些属性标记出来，然后在特定场景下使用的用例。其他方法都不好用，如维护一个可能被修改的列表、依赖时不时会失败的试探，或使用某种严格的命名约定。

下一章将介绍 ES6 的更多特性，包括如何使用 Promise 和生成器进行流程控制，以及如何迭代 JavaScript 对象。

第 4 章

迭代与流程控制

第 2 章介绍了 ES6 中的基础特性，第 3 章讨论了符号，现在我们可以充满信心地来看看 Promise、迭代器和生成器了。Promise 是一种控制异步代码流的新方式。迭代器定义了如何迭代对象，并通过迭代生成值序列。生成器可以用于编写看似同步实则在后台异步工作的代码。这些都是本章即将介绍的内容。

本章先从 Promise 讲起。从 ES6 开始，Promise 成为了 JavaScript 原生支持的特性。

4.1 Promise

Promise 可以大致理解为“保存着一个未来可用的值的代理”。虽然可以在 Promise 中编写同步代码，但 Promise 本身是严格按照异步方式执行的。掌握了 Promise，就可以轻轻松松地实现异步逻辑。

4.1.1 快速理解Promise

为了理解 Promise，我们先来看一个在浏览器中使用 `fetch` API 的示例。`fetch` 是 `XMLHttpRequest` 的简化版，旨在极大简化发送 HTTP 请求的操作。此外它还提供了可扩展的 API，以满足一些复杂场景的应用，但这并不是这里的重点。简单来说，我们可以使用以下代码发送一个 `GET /items` HTTP 请求。

```
fetch('/items')
```

这条 fetch('items') 语句看起来没什么，它只会通过 GET 方法请求 /items 资源，然后就什么也不管了。也就是说，无论请求是否成功，都不会响应。重点是，fetch 方法返回的是一个 Promise！ 因此，我们可以在这个 Promise 后面调用 .then 方法，等 /items 资源加载完成后执行一个回调，并给回调传入响应对象 response 作为参数。

```
fetch('/items').then(response => {
  // 处理响应
})
```

以下代码显示浏览器已经基于 Promise API 实现了 fetch。调用 fetch 会返回一个 Promise 对象。与事件处理相似，在 Promise 中使用 .then 和 .catch 也可以绑定若干个反应函数（reaction）。

```
const p = fetch('/items')
p.then(res => {
  // 处理响应
})
p.catch(err => {
  // 处理错误
})
```

传入 .then 的反应函数用于处理成功兑现的 Promise，通常是一个值；传入 .catch 的反应函数会接收一个拒绝理由 reason，用于处理被拒绝的 Promise。实际上，我们也可以直接将处理拒绝的反应函数作为第二个参数注册到 .then 方法中，其表达的意思与前面的代码一样，如下所示。

```
const p = fetch('/items')
p.then(
  res => {
    // 处理响应
  },
  err => {
    // 处理错误
  }
)
```

另一种写法是省略 .then(fulfillment， rejection) 中的兑现反应函数，这与调用 .then 时省略拒绝反应函数类似。.then(null， rejection) 等价于 .catch(rejection)，如下所示。

```
const p = fetch('/items')
p.then(res => {
  // 处理响应
})
p.then(null, err => {
  // 处理错误
})
```

用 Promise 代替回调和事件

过去，在 Promise 出现之前，JavaScript 一直依赖回调实现链式调用。如果前面示例中的 fetch 要求传入回调，那必须给它提供一个 fetch 操作完成后调用的函数。Node.js 给采用回调形式的异步代码树立了典范，即将传给回调函数的第一个参数保留给异步操作期间抛出的错误对象（可能有，也可能没有）。后面的参数可以用于读取异步操作的结果。一般只使用一个数据参数。如果 fetch 支持回调 API，则可以按照以下代码这样写。

```
fetch('/items', (err, res) => {
  if (err) {
    // 处理错误
  } else {
    // 处理响应
  }
})
```

如果还没有取得 /items 资源，或者 fetch 操作没有触发错误，那么这个回调就不会执行。回调的执行是异步的、非阻塞的。注意，这种方式下只能指定一个回调。这个回调要负责实现接收响应后的所有逻辑，这些事情就留给使用者了。

除了传统的回调，另一种 API 设计选择是事件驱动模型。这种模型下，我们可以在 fetch 返回的对象上注册不同事件的回调，每个事件都可以注册任意数量的事件处理器，这与在浏览器 DOM 中给元素添加事件监听器类似。遇到错误时通常会触发一个 error 事件；在相应的重要时刻还会触发其他事件。以下代码展示了支持基于事件的 API 的 fetch。

```
fetch('/items')
  .on('error', err => {
    // 处理错误
  })
  .on('data', res => {
    // 处理响应
  })
```

为不同事件绑定多个监听器，可以解决前面在一个回调函数中集中处理所有后续逻辑的问题。然而，事件的回调函数不方便连缀起来或在另一个异步任务兑现时触发，因此才有了 Promise。不仅如此，事件更适合处理值流，不太适合我们讨论的场景。9.7 节将深入探讨异步代码流程设计，包括哪种结构适合采用哪种代码流。

说到 Promise，连缀调用应该是最难理解的。在基于事件的 API 中，使用 .on 方法添加事件监听器，然后再返回事件发射器（emitter）本身，就可以实现连续注册。Promise 不一样，它的 .then 和 .catch 方法每次都返回一个新的 Promise 对象。记住这一点非常重要，连缀调用的结果会因 .then 和 .catch 添加的位置而迥然不同！

可视化 Promise 链：困惑之源

`.then` 和 `.catch` 方法每次都返回一个新的 Promise，以创建一个类似树形的结构。如果有两个 Promise——`p1` 和 `p1.then` 返回的 `p2`，则 `p1` 和 `p2` 这两个节点就是通过 `p1.then` 的反应函数连接的。反应函数负责创建新的 Promise，并将其以自己反应的 Promise 子节点的身份添加到树中。

连缀 Promise 时，需要理解 `p1.then(r1).then(r2)` 会创建两个新的 Promise：`p2` 和 `p3`。第二个反应函数 `r2` 会在 `p2` 兑现时触发，而 `r1` 会在 `p1` 兑现时触发。如果前面语句的写法是 `p1.then(r1); p1.then(r2)`，那么 `r1` 和 `r2` 在 `p1` 兑现时会同时触发。如果 `p1` 兑现而 `p2` 未兑现，则会出现问题。

厘清 Promise 这个类似树的结构链是理解其各种行为的关键。为此，我创建了一个在线工具 Promisees，用于可视化地展示 Promise 生成的树结构，如图 4-1 所示。



图 4-1：Promisees 可以显示代码中的 Promise 在兑现或拒绝之后的树形结构

创建 Promise 需要向 `Promise` 构造函数传入一个解决函数（resolver），该函数接收 `resolve` 和 `reject` 两个参数，用于决定何时以及如何处理创建的 Promise。如果需要兑现，则调用 `resolve` 方法；如果需要拒绝，则调用 `reject` 方法。在调用这两个方法解决 Promise 之前，Promise 处于待定状态，此时任何反应函数都不会执行。以下代码创建了一个新的 Promise，一秒后再根据随机值决定以兑现还是拒绝来解决它。

```
new Promise(function (resolve, reject) {
  setTimeout(function () {
    if (Math.random() > 0.5) {
      resolve('random success')
    } else {
      reject(new Error('random failure'))
    }
  }, 1000)
})
```

还可以使用 `Promise.resolve` 和 `Promise.reject` 来创建 Promise。这两个方法创建的 Promise

会分别以一个兑现的值和一个拒绝理由立即解决。

```
Promise
  .resolve({ result: 123 })
  .then(data => console.log(data.result))
// <- 123
```

如果 Promise p 兑现了，那么通过 p.then 注册的反应函数会执行。如果 p 被拒绝了，那么通过 p.catch 注册的反应函数会执行。根据反应函数是返回一个值、一个 Promise、一个 Thenable，还是抛出一个错误，执行反应函数会相应地产生三种结果。有关 Thenable 的内容参见 4.1.3 节，它指的是可以通过 Promise.resolve 转换为 Promise 的包含 then 方法的对象。

如果反应函数返回一个值，则 .then 返回的 Promise 会以该值兑现。这种情况下，多个 Promise 可以连缀起来对前面 Promise 兑现的值一步步进行转换，如下所示。

```
Promise
  .resolve(2)
  .then(x => x * 7)
  .then(x => x - 3)
  .then(x => console.log(x))
// <- 11
```

如果反应函数返回一个 Promise，则以下代码中的第一个 .then 返回的 Promise 会被阻塞，直到它的反应函数返回的 Promise 兑现：由于调用了 setTimeout，至少要等两秒。

```
Promise
  .resolve(2)
  .then(x => new Promise(function (resolve) {
    setTimeout(() => resolve(x * 1000), x * 1000)
  }))
  .then(x => console.log(x))
// <- 2000
```

如果反应函数抛出一个错误，那么会导致 .then 返回的 Promise 被拒绝，进而转向 .catch 分支，以抛出的错误对象作为拒绝理由。以下代码为 fetch 添加了一个兑现反应函数。一旦 fetch 操作兑现，它就会抛出一个错误，导致注册到 .then 返回的 Promise 上的拒绝反应函数执行。

```
const p = fetch('/items')
  .then(res => { throw new Error('unexpectedly') })
  .catch(err => console.error(err))
```

接下来我们后退一步并放慢脚步，多看几个使用 Promise 的示例。

4.1.2　Promise的延续与连缀

上一节提到过，每个 .then 都会返回自己的新 Promise，因此可以连缀无数个 .then。那么这个过程中到底发生了什么？如何正确推断 Promise 的执行？发生错误时怎么办？

当一个 Promise 的解决函数中发生错误时，我们可以用 p.catch 来捕获该错误，如下所示。

```
new Promise((resolve, reject) => reject(new Error('oops')))
  .catch(err => console.error(err))
```

可见，如果解决函数调用了 reject，那么 Promise 会以拒绝来解决。如果解决函数中抛出了一个异常，那么结果也一样，如下所示。

```
new Promise((resolve, reject) => { throw new Error('oops') })
  .catch(err => console.error(err))
```

兑现和拒绝反应函数中发生的错误会导致相同的结果，即有一个 Promise 会被拒绝。是哪个 Promise 呢？就是以发生错误的反应函数为参数的 .then 或 .catch 返回的那个 Promise。通过代码很容易说明这一点。

```
Promise
  .resolve(2)
  .then(x => { throw new Error('failed') })
  .catch(err => console.error(err))
```

将前面代码中每一步的方法调用拆解为变量会更易理解。以下代码说明，如果将 .catch 添加给 p1，则无法捕获 .then 反应函数抛出的错误。虽然 p1 兑现了，但 p2（调用 p1.then 得到的另一个 Promise）因抛出的错误被拒绝了。只有将拒绝反应函数通过 .catch 添加到 p2，才能捕获这个错误。

```
const p1 = Promise.resolve(2)
const p2 = p1.then(x => { throw new Error('failed') })
const p3 = p2.catch(err => console.error(err))
```

图 4-2 可视化地展示了树形的 Promise 调用结构。可以看出，在 p2 节点出错的情况下，添加给 p1 的拒绝反应函数得不到消息。

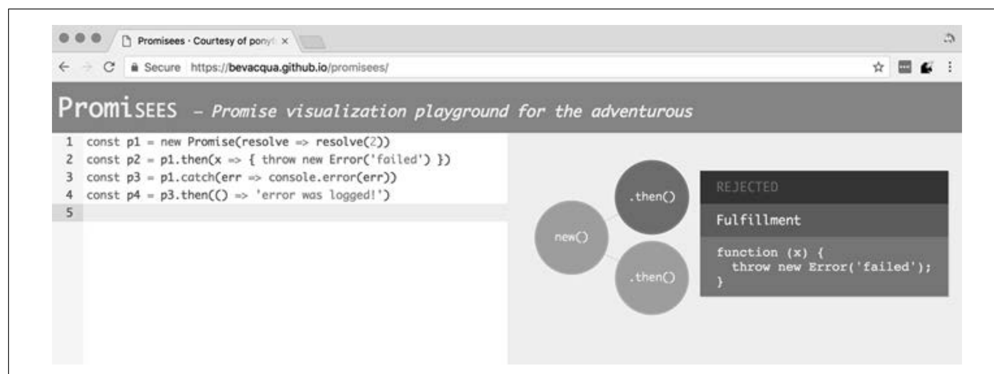


图 4-2：根据 Promise 连缀调用的树形结构可知，拒绝反应函数只能捕获基于 Promise 的代码分支上的错误

要想处理 p2 的错误，必须将反应函数注册到 p2，如图 4-3 所示。

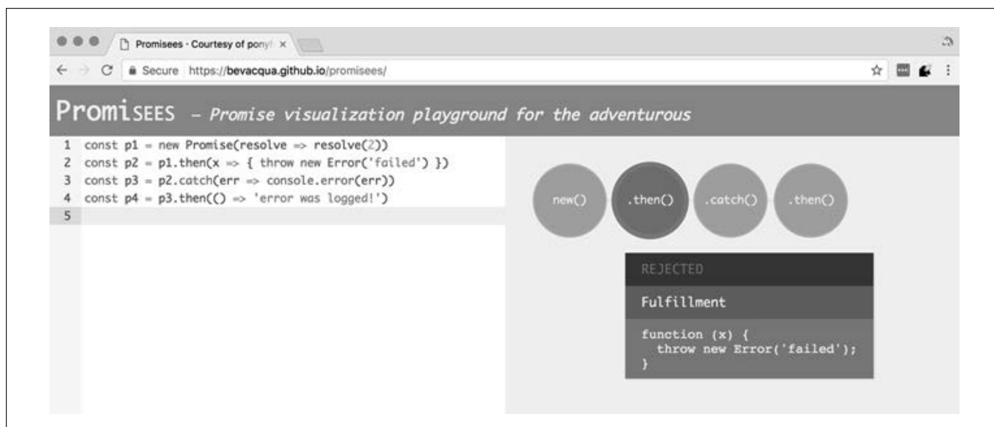


图 4-3：将拒绝处理器注册到发生错误的分支就可以捕获错误了

现在我们知道了，在注册反应函数时，搞清楚应该注册到哪个 Promise 很重要，因为这关系到最终能否捕获错误。需要注意的另一点是，只要 Promise 链中有未捕获的错误，那么下游的拒绝处理器总能捕获到它。以下示例在发生错误的 p2 和注册拒绝反应函数的 p4 之间插入了一个 .then 调用。在 p2 因拒绝而解决后，p3 也会因拒绝而解决，因为它直接依赖 p2。当 p3 因拒绝而解决时，p4 上注册的拒绝处理器就会执行。

```
const p1 = Promise.resolve(2)
const p2 = p1.then(x => { throw new Error('failed') })
const p3 = p2.then(x => x * 2)
const p4 = p3.catch(err => console.error(err))
```

一般来说，此时的 p4 会兑现，因为 .catch 拒绝处理器中并未抛出错误。这意味着，如果 p4.then 注册了兑现处理器，那么该函数就会继续执行。以下示例表明，在 p3 成功解决后，依赖它的 p4 注册的兑现处理器会执行，并在浏览器控制台输出相关信息。

```
const p1 = Promise.resolve(2)
const p2 = p1.then(x => { throw new Error('failed') })
const p3 = p2.catch(err => console.error(err))
const p4 = p3.then(() => console.log('crisis averted'))
```

同样，如果 p3 拒绝处理器中发生错误，我们也可以用 .catch 来捕获它。以下示例展示了如何通过 p3.catch 捕获被拒绝的 p3 抛出的异常。

```
const p1 = Promise.resolve(2)
const p2 = p1.then(x => { throw new Error('failed') })
const p3 = p2.catch(err => { throw new Error('oops') })
const p4 = p3.catch(err => console.error(err))
```

以下代码会打印 err.message 一次，而不是两次。因为第一个 .catch 中并未发生错误，所以它后面的拒绝分支自然不会执行。

```
fetch('/items')
  .then(res => res.a.prop.that.does.not.exist)
  .catch(err => console.error(err.message))
  .catch(err => console.error(err.message))
// <- 'Cannot read property "prop" of undefined'
```

以下代码倒是会打印 err.message 两次。这是因为我们将 .then 返回的 Promise 保存到了变量 p 中，然后给 p 添加了两个 .catch 反应函数。前面代码中的第二个 .catch 会捕获第一个 .catch 返回的 Promise 中的错误，而以下示例中的两个 .catch 拒绝处理器响应的是同一个 Promise。

```
const p = fetch('/items').then(res =>
  res.a.prop.that.does.not.exist
)
p.catch(err => console.error(err.message))
p.catch(err => console.error(err.message))
// <- 'Cannot read property "prop" of undefined'
// <- 'Cannot read property "prop" of undefined'
```

可以看到，Promise 是可以任意连缀的。正如前面看到的，我们可以在 Promise 链的任何一步保存一个引用，然后在其基础上连缀更多的 Promise。这也是理解 Promise 时的重要一点。

接下来我们以下面的代码为例，逐步分析创建和连缀 Promise 的过程中发生的事情。我们先认真看看以下代码。

```
const p1 = fetch('/items')
const p2 = p1.then(res => res.a.prop.that.does.not.exist)
const p3 = p2.catch(err => {})
const p4 = p3.catch(err => console.error(err.message))
```

以下列出了前面代码执行时发生了什么。

(1) fetch 返回一个新的 Promise p1。
(2) 如果 p1 兑现，则 p1.then 返回一个新的 Promise p2。
(3) 如果 p2 拒绝，则 p2.catch 返回一个新的 Promise p3。
(4) 如果 p3 拒绝，则 p3.catch 返回一个新的 Promise p4。
(5) p1 兑现，p1.then 反应函数执行。
(6) 接着，由于 p1.then 反应函数中发生错误，p2 拒绝。
(7) 因为 p2 拒绝，所以 p2.catch 执行，并忽略 p2.then 分支（如果有的话）。
(8) 因为 p2.catch 没有发生错误或返回拒绝的 Promise，所以它返回的 p3 兑现。
(9) 因为 p3 兑现，所以 p3.catch 会被忽略，p3.then 分支会执行（如果有的话）。

我们应该将 Promise 的调用链想象成树形结构。再次强调：应该将 Promise 的调用链想象成树形结构[1]。图 4-4 可以帮助我们强化这个认知。

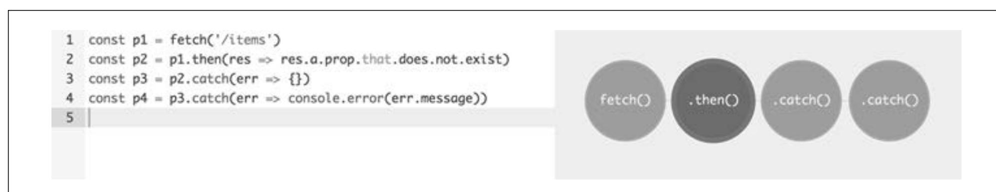


图 4-4：在这个树结构中，p3 兑现了，因为它没有抛出异常，也没有被拒绝。为此，p4 永远不会走向拒绝分支，因为其节点兑现了

这些最初都是从一个 Promise 开始的，接下来我们将探讨如何创建 Promise。然后给 Promise 添加 .then 或 .catch 分支，再给每个分支添加任意数量的 .then 或 .catch，以创建新分支。

4.1.3 创建Promise

我们已经知道，可以通过 fetch、Promise.resolve、Promise.reject 或 Promise 构造函数创建 Promise。前面的示例主要使用 fetch。接下来我们看看其他三种方式。

我们可以使用 new Promise(resolver) 创建 Promise，其中 resolver 参数是一个函数，用于解决新创建的 Promise。resolver 会接收两个均为函数的参数：一个叫 resolve，另一个叫 reject。

以下代码创建的两个 Promise 分别以兑现和拒绝得到解决。第一个 Promise 兑现时使用了值 'result'，第二个 Promise 由一个 Error 对象拒绝，错误消息为 'reason'，即拒绝的理由。

```
new Promise(resolve => resolve('result'))
new Promise((resolve, reject) => reject(new Error('reason')))
```

兑现或拒绝 Promise 时不返回值也是可以的，但没有什么意义。一般来说，兑现 Promise 时都要返回一个结果，比如 Ajax 调用的响应，就像前面使用 fetch 时那样。如果拒绝 Promise，那一定要给出拒绝的理由，通常要将这个理由包装在 Error 对象中，以便上报错误，方便排错。

你可能已经猜到了，Promise 的解决函数并不要求代码必须同步。在解决函数中，兑现和拒绝 Promise 都可以是异步的。即使解决函数立即调用 resolve，返回的结果也要等到下一个事件循环才会传到注册的反应函数。这就是 Promise 最关键的地方！以下代码中的 Promise 会在 2 秒后兑现。

```
new Promise(resolve => setTimeout(resolve, 2000))
```

注 1：我编写了一个名为 Promisees 的在线可视化工具，你可以在其中的 Promise 链下看到树结构。

注意，只有这些函数（resolve 或 reject）的第一次调用才有作用。一旦解决，Promise 的结果就不会再改变了。以下代码创建了一个 Promise，它要么在 delay 时间到时兑现，要么在 3 秒后被拒绝。这里利用了解决 Promise 后再调用任何函数都不起作用的事实创造了竞争条件，对任意函数的第一次调用决定结果。

```
function resolveUnderThreeSeconds(delay) {
  return new Promise(function (resolve, reject) {
    setTimeout(resolve, delay)
    setTimeout(reject, 3000)
  })
}
resolveUnderThreeSeconds(2000) // 2秒后兑现
resolveUnderThreeSeconds(7000) // 3秒后拒绝
```

在创建新 Promise p1 时，除了可以给 resolve 传递非 Promise 值，还可以给它传递另一个 Promise p2。这种情况下，p1 要等到 p2 解决后才能解决。只要 p2 一解决，p1 就会用它的值和输出进行解决。因此，以下代码与简单地调用 fetch('/items') 效果相同。

```
new Promise(resolve => resolve(fetch('/items')))
```

但这种行为仅限于使用 resolve。如果想要通过 reject 复现相同的行为，你会发现 p1 被拒绝，拒绝理由就是 p2。虽然 resolve 可以导致 Promise 兑现或拒绝，但 reject 只能导致 Promise 被拒绝。如果传给 resolve 的 Promise 被拒绝或最终会被拒绝，那么创建的 Promise 也会被拒绝。反过来对 reject 并不适用。如果在解决函数中调用 reject，那么无论传给 reject 的是什么值，你最终都会得到一个状态为拒绝的 Promise。

在一些情况下，你会提前知道解决 Promise 要用到哪个值。此时你可以直接创建一个 Promise，如下所示。接下来你就可以用 Promise 链实现自己的逻辑，而不必借助返回某些 Promise 的调用（如 fetch 的调用）。

```
new Promise(resolve => resolve(12))
```

当然，如果你只是想要一个预先解决的 Promise，这样还略显冗长。此时可以使用更简短的 Promise.resolve。以下代码与前面示例的结果相同。如果说有区别，也只是语义上的。但这样可以省去声明解决函数的麻烦，而且语法也更方便 Promise 延续及连缀，看起来也更好理解。

```
Promise.resolve(12)
```

与前面看到的 resolve(fetch) 的情况类似，我们也可以使用 Promise.resolve 来封装其他 Promise 或者将一个 Thenable 对象转换为相应的 Promise。以下代码展示了如何用 Promise.resolve 将一个 Thenable 对象转换成 Promise，然后像使用其他 Promise 一样使用它。

```
Promise
  .resolve({ then: resolve => resolve(12) })
  .then(x => console.log(x))
// <- 12
```

如果提前知道了拒绝的理由，那么你可以使用 `Promise.reject`。以下代码展示了如何使用已知的理由 `reason` 来创建最终会被拒绝的 Promise。我们可以在反应函数中使用 `Promise.reject` 来代替 `throw` 语句。`Promise.reject` 的另一个用途是作为箭头函数的隐式返回值，这是使用 `throw` 语句无法做到的。

```
Promise.reject(reason)
fetch('/items').then(() =>
  Promise.reject(new Error('arbitrarily'))
)
fetch('/items').then(() => { throw new Error('arbitrarily')})
```

可以想见，你在实践中不会使用太多 `new Promise`。这个构造函数更多地会被支持 Promise 或原生函数（如 `fetch`）的库在内部使用。既然通过连缀 `.then` 和 `.catch` 可以创建树形调用结构，在原始 Promise 的基础上任意扩展，那么在 API 代码的开头调用一次 `new Promise` 基本就足够了。无论如何，理解 Promise 的创建方式对于掌握基于 Promise 的控制流至关重要。

4.1.4　Promise的状态

Promise 有三种可能的状态：待定、兑现和拒绝。默认状态为待定，之后可能转换为兑现，也可能转换为拒绝。

Promise 只能解决或拒绝一次。对已确定的 Promise 再兑现或再拒绝不会产生任何效果。

如果 Promise 解决为一个非 Promise、非 `Thenable` 对象，则确定状态为兑现。如果 Promise 被拒绝，那它的状态也是确定的，只不过状态为拒绝。

如果 Promise `p1` 解决为另一个 Promise 或 `Thenable` `p2`，那么 `p1` 会处于待定状态，但最终会解决（然后就不能再解决或拒绝了）。`p2` 确定后，其输出会转到 `p1`，`p1` 也会随之确定。

Promise 一旦兑现，通过 `p.then` 注册的反应函数就会尽快执行。同样，Promise 被拒绝后，通过 `p.catch` 注册的反应函数就会尽快执行。Promise 确定之后添加的反应函数也会尽快执行。

接下来我们看一个人为设计的示例，它说明可以在第一个 `fetch` 请求的 `.then` 反应函数中创建第二个 `fetch` Promise。第二个 `fetch` 请求能且只能在第一个 Promise 兑现后发出。`console.log` 语句能且只能在第二个 Promise 兑现后才在控制台中打印 `done`。

```
fetch('/items')
  .then(() => fetch('/item/first'))
  .then(() => console.log('done'))
```

现在我们再来看一个贴近实际的示例，其中涉及更多步骤。这个例子要先取得第一个 fetch 请求的输出，然后根据该输出创建第二个请求。为此，这里使用 res.json 方法来返回一个解决为 JSON 对象的 Promise。接着用这个对象拼接出第二个 fetch 请求的地址。最后，将第二个请求取得的 item 对象打印到控制台。

```
fetch('/items')
  .then(res => res.json())
  .then(items => fetch(`/item/${ items[0].slug }`))
  .then(res => res.json())
  .then(item => console.log(item))
```

返回的结果并不局限于 Promise 或 Thenable。.then 或 .catch 反应函数也可以返回具体的值。这些值会沿 Promise 链传给下一个反应函数。因此，我们可以将反应函数看作 Promise 链中前一个反应函数的输入到下一个反应函数的输入的转换装置。以下示例先创建了一个兑现值为 [1，2，3] 的 Promise，后面的反应函数会将这些值转换为 [2，4，6]，最后一个反应函数会在控制台中输出这些值。

```
Promise
  .resolve([1, 2, 3])
  .then(values => values.map(value => value * 2))
  .then(values => console.log(values))
  // <- [2, 4, 6]
```

注意，我们也可以转换拒绝分支中的数据。4.1.3 节中介绍过，如果 .catch 反应函数执行完没有出现错误，且没有返回被拒绝的 Promise，那么它返回的就是兑现的 Promise，因此接下来会走 .then 分支。

4.1.5 Promise#finally提案

有人向 TC39 提交了一个有关 Promise#finally 方法的提案[2]，即在 Promise 确定之后，无论是确定兑现还是确定拒绝，都会调用一个反应函数。

我们可以将以下代码看作 Promise#finally 的简单替代。也就是说，将一个回调传入 p.then，既用作兑现反应函数，也用作拒绝反应函数。

```
function finally(p, fn) {
  return p.then(
    fn,
    fn
  )
}
```

但是二者在语义上有差别。首先，传给 Promise#finally 的反应函数不接收任何参数，因

注 2：这个建议已经在 2018 年 1 月 TC39 第 62 次会议上表决通过为阶段 4（完成）。——译者注

为此时的 Promise 既可能是兑现的，也可能是被拒绝的。对用户而言，`Promise#finally` 一般用于隐藏 `fetch` 请求执行时显示的加载动画，或执行其他清理任务，此时并不需要知道 Promise 确定后的值。经过修改，以下代码没有向任何反应函数传递参数。

```
function finally(p, fn) {
  return p.then(
    () => fn(),
    () => fn()
  )
}
```

传给 `Promise#finally` 的反应函数会解决为父 Promise 的结果。

```
const p1 = Promise.resolve('value')
const p2 = p1.finally(() => {})
const p3 = p2.then(data => console.log(data))
// <- 'value'
```

这一点与 `p.then(fn, fn)` 不同。除非在反应函数中显式转发父 Promise 的值，否则 `p.then(fn, fn)` 会生成一个新的兑现值，如下所示。

```
const p1 = Promise.resolve('value')
const p2 = p1.then(() => {}, () => {})
const p3 = p2.then(data => console.log(data))
// <- undefined
```

以下代码给出了 `Promise#finally` 的完整替代脚本。

```
function finally(p, fn) {
  return p.then(
    result => resolve(fn()).then(() => result),
    err => resolve(fn()).then(() => Promise.reject(err))
  )
}
```

注意，如果传给 `Promise#finally` 的反应函数被拒绝或抛出错误，那么 `Promise#finally` 返回的 Promise 也会以相同的原因被拒绝，如下所示。

```
const p1 = Promise.resolve('value')
const p2 = p1.finally(() => Promise.reject('oops'))
const p3 = p2.catch(err => console.log(err))
// <- 'oops'
```

仔细查看替代脚本后你会发现，如果两个反应函数中的任何一个出现异常，那么传入的 Promise 就会被拒绝。此时，通过 `Promise.reject` 或其他方式返回被拒绝的 Promise 意味着 `resolve(fn())` 导致 Promise 被拒绝，因而调用 `.finally` 的用于返回 Promise 确定值的 `.then` 分支（如果有的话）将不会执行。

4.1.6 Promise.all和Promise.race

编写异步代码时，可能两个任务中的一个任务会依赖另一个任务的结果，因此这两个任务必须串行执行。有时两个任务也可能互相没有依赖，因此可以并行执行。当然，Promise 很擅长控制异步顺序流，因为一个 Promise 就足以触发一个事件链，这些事件会相继发生。Promise 还通过另外两个 API 方法为执行并行任务提供了解决方案：`Promise.all` 和 `Promise.race`。

多数情况下，我们都应该使用这两个 API 来执行并行任务，因为它们速度很快。假设我们想要取得库存目录中两个商品的简介，那么可以使用两个 API 调用，然后将结果打印到控制台。以下代码可以同时运行两个操作，但必须分别打印结果。对这两个任务来说，将结果打印到控制台的操作没什么区别。如果我们想要只调用一次 API，但同时传入两个商品，那就不能使用两个 `fetch` 请求。

```
fetch('/products/chair')
  .then(r => r.json())
  .then(p => console.log(p))
fetch('/products/table')
  .then(r => r.json())
  .then(p => console.log(p))
```

`Promise.all` 方法接收一个 Promise 数组，并返回一个 Promise，比如 `p`。在传给 `Promise.all` 的所有 Promise 都兑现后，`p` 也会随之兑现并按照传入顺序返回各 Promise 兑现的结果数组。但只要有一个 Promise 被拒绝，那么 `p` 就会立即以该 Promise 的拒绝理由确定为被拒绝。以下代码用 `Promise.all` 获取两个商品的信息，然后只用一条 `console.log` 语句打印结果。

```
Promise
  .all([
    fetch('/products/chair'),
    fetch('/products/table')
  ])
  .then(products => console.log(products[0], products[1]))
```

因为返回结果是一个数组，所以代码中用到了索引，但这里的索引没有什么语义上的价值。为此，我们可以使用参数解构将每个商品的信息赋给相应的变量，这样更容易让人理解代码的含义。以下示例用解构提升了代码的可读性。注意，即使只有一个参数，但由于要使用解构，这里也必须用括号将箭头函数的参数声明括起来。

```
Promise
  .all([
    fetch('/products/chair'),
    fetch('/products/table')
  ])
  .then(([chair, table]) => console.log(chair, table))
```

以下代码表明，如果一个 Promise 被拒绝，那么最终的 `p` 也会被拒绝。理解这一点很重要：只要有一个 Promise 被拒绝，那么结果数组就不是完全兑现的，因而 `p` 就无法兑现。在以下示例中，由于 `p1` 率先被拒绝，`p` 也会马上被拒绝，不会再等 `p2` 和 `p3` 的解决结果。

```
const p1 = Promise.reject('failed')
const p2 = fetch('/products/chair')
const p3 = fetch('/products/table')
const p = Promise
  .all([p1, p2, p3])
  .catch(err => console.log(err))
  // <- 'failed'
```

总结一下，`Promise.all` 可能有以下三种结果。

- 所有传入的 Promise 都兑现，因此结果 Promise 也兑现。
- 只要传入的 Promise 有一个被拒绝，那么结果 Promise 也会被拒绝。
- 如果依赖的 Promise 一个都没有被拒绝，且至少还有一个 Promise 处于待定状态，那么结果 Promise 也会处于待定状态。

`Promise.race` 方法与 `Promise.all` 接收的参数一样，区别在于前者中确定状态的第一个 Promise 将“赢得比赛”，其结果会决定 `Promise.race` 返回的 Promise 的状态。

```
Promise
  .race([
    new Promise(resolve => setTimeout(() => resolve(1), 1000)),
    new Promise(resolve => setTimeout(() => resolve(2), 2000))
  ])
  .then(result => console.log(result))
  // <- 1
```

拒绝状态同样会导致“比赛结束”，并且结果 Promise 的状态也是被拒绝。`Promise.race` 可用于对无法控制的 Promise 设置超时。例如，以下示例中传给 `Promise.race` 的两个 Promise 一个是 `fetch` 请求返回的，另一个会在 5 秒后被拒绝。因此，如果请求时间超过 5 秒，则“比赛”结果就是拿到一个被拒绝的 Promise。

```
function timeout(delay) {
  return new Promise(function (resolve, reject) {
    setTimeout(() => reject('timeout'), delay)
  })
}
Promise
  .race([
    fetch('/large-resource-download'),
    timeout(5000)
  ])
  .then(res => console.log(res))
  .catch(err => console.log(err))
```

4.2 迭代器协议与可迭代协议

ES6 中引入了两个新协议：迭代器和可迭代对象。这两个协议可以为任何对象定义迭代行为。本节将先介绍如何将对象转换成可迭代序列。之后会说明怎样才可以偷懒，即用迭代器定义无穷序列。最后，讨论定义可迭代对象时的一些现实考量。

4.2.1 迭代的原理

普通对象要想转换成可迭代对象，必须遵守一个协议：给这个对象的 `Symbol.iterator` 属性赋值一个函数。如果这个对象需要迭代，那么每次迭代都会调用赋给 `Symbol.iterator` 的可迭代协议方法。

第 2 章介绍过扩展操作符，它是 ES6 引入的利用可迭代协议的几个新语言特性之一。对以下假想的 `iterable` 对象使用扩展操作符时就会查询这个对象的 `Symbol.iterator` 方法，因为这个对象实现了迭代器协议。返回的迭代器将用于获取结果的值。

```
const sequence = [...iterable]
```

前面说过，符号属性不能直接嵌入对象字面量的键中。以下代码展示了如何使用 ES6 之前的语法给对象添加符号属性。

```
const example = {}
example[Symbol.iterator] = fn
```

在 ES6 中，我们可以使用计算属性名将符号键加到对象字面量中，避免像前面那样多写一行代码，如下所示。

```
const example = {
  [Symbol.iterator]: fn
}
```

赋值给 `Symbol.iterator` 的方法必须返回一个对象，该对象必须遵守迭代器协议。这个协议规定了如何从可迭代序列中取值。根据协议，迭代器方法返回的对象必须有一个 `next` 方法。`next` 方法不接受参数，并且返回一个包含以下两个属性的对象。

- `value`，序列中的当前值。
- `done`，布尔值，表明序列是否结束。

接下来我们通过一个示例来理解迭代协议背后的概念。通过添加 `Symbol.iterator` 属性，我们将 `sequence` 对象写成了一个可迭代对象。这个可迭代对象（的 `Symbol.iterator` 方法）返回一个迭代器对象。每次调用 `next` 取序列中的下一个值时，都会返回 `items` 数组中的一个元素。当 `i` 大于 `items` 数组的最大索引值时，则返回 `done: true`，以表明序列已经迭代完毕。

```
const items = ['i', 't', 'e', 'r', 'a', 'b', 'l', 'e']
const sequence = {
  [Symbol.iterator]() {
    let i = 0
    return {
      next() {
        const value = items[i]
        i++
        const done = i > items.length -1
        return { value, done }
      }
    }
  }
}
```

JavaScript 是一门持续进化的语言，新增特性永远不会破坏原有代码。因此，可迭代对象不能利用已有的 `forEach` 和 `for..in` 进行迭代。ES6 为迭代可迭代对象提供了以下几种方式：`for..of`、扩展操作符 `...` 和 `Array.from`。

`for..of` 迭代方法可用于迭代任何可迭代对象。以下示例展示了如何通过它迭代前面定义的可迭代对象 `sequence`。

```
for (const item of sequence) {
  console.log(item)
  // <- 'i'
  // <- 't'
  // <- 'e'
  // <- 'r'
  // <- 'a'
  // <- 'b'
  // <- 'l'
  // <- 'e'
}
```

常规对象可以像刚刚介绍的那样通过实现 `Symbol.iterator` 变成可迭代对象。在 ES6 的框架下，`Array`、`String`、DOM 中的 `NodeList` 以及 `arguments` 默认都是可迭代的，这也为 `for..of` 提供了用武之地。要想从值的可迭代序列获取一个数组，可以使用扩展操作符；扩展后的每一项都会成为结果数组中的一个元素。使用 `Array.from` 也可以实现相同的效果。此外，`Array.from` 还可以将类似数组的对象（具有 `length` 属性，属性是以 0 开头的整数）转换成数组。

```
console.log([...sequence])
// <- ['i', 't', 'e', 'r', 'a', 'b', 'l', 'e']
console.log(Array.from(sequence))
// <- ['i', 't', 'e', 'r', 'a', 'b', 'l', 'e']
console.log(Array.from({ 0: 'a', 1: 'b', 2: 'c', length: 3 }))
// <- ['a', 'b', 'c']
```

我们借 `sequence` 对象来总结一下。`sequence` 对象遵守了可迭代协议，因为它给 `[Symbol.`

iterator] 赋值了一个方法。结果 sequence 就成了可迭代对象，即它可以被迭代。这里说的方法返回一个对象，该对象遵守迭代器协议。当我们需要迭代对象时，迭代器方法就会被调用，而它返回的迭代器则用于从 sequence 对象中取值。要想迭代可迭代对象，可以使用 for..of、扩展操作符或 Array.from 方法。

从本质上来说，这两个协议的优势在于它们提供了有意义的方式，让我们能够轻松地迭代集合和类似数组的对象。将任意对象定义为可迭代对象的功能非常强大，这样不同的库就可以因遵循语言原生支持的协议而具有共性：可以迭代。如前所示，实现迭代器协议一点也不麻烦，而且由于这是新增特性，也不会影响原有代码的行为。

例如，jQuery 及 document.querySelectorAll 都返回类似数组的对象。如果 jQuery 在其集合原型上实现了迭代器协议，那么就可以使用原生的 for..of 来迭代其集合中的元素。

```
for (const element of $('li')) {
  console.log(element)
  // <- jQuery集合中的<li>元素
}
```

可迭代对象不一定是有限的，其元素也可以是无限的。下面就来讨论一下无穷序列及其实现。

4.2.2 无穷序列

迭代器本质上是懒惰的。迭代器序列中的元素每次只能取出一个，即使序列是有限的。注意，如果没有懒惰的属性，那么无穷序列甚至无法表示。无穷序列不能用数组表示，否则用扩展操作符或 Array.from 将序列转换成数组会导致 JavaScript 引擎崩溃，因为这会进入无穷循环。

以下代码展示了一个生成 0~1 范围内随机浮点数的无穷序列的迭代器。注意，next 方法返回的对象中根本没有值为 true（表示序列结束）的 done 属性。代码使用两个箭头函数隐式返回了两个对象。第一个返回的是迭代器对象，用于遍历随机数组成的无穷序列。第二个箭头函数用 Math.random() 从序列中取出每个值。

```
const random = {
  [Symbol.iterator]: () => ({
    next: () => ({ value: Math.random() })
  })
}
```

使用 Array.from(random) 或 [...random] 将可迭代对象 random 转换为数组均会导致程序崩溃，因为这个序列没有尽头。无穷序列很容易将浏览器、Node.js 服务器进程搞崩溃，因此要小心使用。

我们可以通过几种方式避免无穷循环的风险，以安全地使用无穷序列。第一种方式是用解构取出序列中特定位置的值，如下所示。

```
const [one, another] = random
console.log(one)
// <- 0.23235511826351285
console.log(another)
// <- 0.28749457537196577
```

解构无穷序列不适合大量取值，特别是要设置动态条件的情况下，比如取出前 `i` 个值或在满足某个条件时一直取值。此时最好使用 `for..of`，以定义中断条件来避免无穷循环。这是以编程方式取得所需值的最好办法。以下示例就用 `for..of` 迭代了无穷序列，但只要有值大于 `0.8`，就中断循环。鉴于 `Math.random` 能生成 0~1 的任意值，循环一定会中断。

```
for (const value of random) {
  if (value > 0.8) {
    break
  }
  console.log(value)
}
```

不过，这样的代码以后再回过头来看会很难理解，因为很多代码都在处理迭代序列、打印随机数，还有一个最大值作为条件。将其中一部分逻辑抽象到另一个方法中可以使代码更好理解。

从无穷序列中取值的一个常见模式是取前几个值。虽然可以用 `for..of` 配合 `break` 达到目的，但最好将它们抽象为一个 `take` 方法。以下示例展示了 `take` 方法的一种实现方式。这个方法接收一个 `sequence` 和一个表示想从 `sequence` 中取前几个值的 `amount` 参数，并返回一个可迭代对象。在迭代这个对象时，它会取得传入序列的迭代器，其 `next` 方法也会代理到传入的序列。只要 `amount` 不小于 1 就会返回值，否则结束序列。

```
function take(sequence, amount) {
  return {
    [Symbol.iterator]() {
      const iterator = sequence[Symbol.iterator]()
      return {
        next() {
          if (amount-- < 1) {
            return { done: true }
          }
          return iterator.next()
        }
      }
    }
  }
}
```

我们的实现对无穷序列非常有效，因为它包含了一个固定的退出条件：只要 amount 小于 1，take 返回的序列就结束。现在不用再像前面那样从 random 中取值了，只要按照以下方式编写代码即可。

```
[...take(random, 2)]
// <- [0.304253100650385, 0.5851333604659885]
```

这个模式可以将任何无穷序列归约为有限序列。如果你想要的有限序列不是“前 N 个值”，而是前面所示的“第一个大于 0.8 的数值之前的所有随机数”，修改一下 take 的退出条件就可以了。以下示例中的 range 函数有一个默认值为 0 的参数 low 和一个默认值为 1 的参数 high。只要序列中有任何值越界，就立即停止取值。

```
function range(sequence, low = 0, high = 1) {
  return {
    [Symbol.iterator]() {
      const iterator = sequence[Symbol.iterator]()
      return {
        next() {
          const item = iterator.next()
          if (item.value < low || item.value > high) {
            return { done: true }
          }
          return item
        }
      }
    }
  }
}
```

现在不用因为害怕无穷循环永无尽头而中断 for..of 循环了，我们可以确保，只要取到的值超出预期范围，循环一定会退出。这样一来，代码关注的就不再是序列如何生成，而是如何使用这个序列。以下示例根本用不着 for..of，因为退出条件已经内置在作为中间层的 range 函数中了。

```
const low = [...range(random, 0, 0.8)]
// <- [0.68912092433311, 0.059788614744320, 0.09396195202134]
```

这种将复杂性抽象到另一个函数中的做法有助于保持代码意图明确，使得我们只想取得一个派生序列时无须使用 for..of 循环。这个示例也表明，序列可以相互融合和嵌入。我们先创建了一个通用的无穷序列 random，然后将它输送到 range 函数，这个函数会返回一个派生的序列，该序列会在取值小于或大于设定范围时终止。有关迭代器的非常重要的一点是，不管有没有组合，range 函数返回的迭代器也可以懒惰迭代。这就意味着你可以组合任意数量的迭代器，对它们进行遍历、过滤和作为退出条件。

> **发现无穷序列**
>
> 迭代器本身没有任何标识来表明它生成的序列是否无穷。与经典的停机问题一样（见图 4-5），有时在代码中无法知道某个序列是否为无穷序列。
>
> 
>
>
> 图 4-5：XKCD 漫画中描述的停机问题
>
> 一般情况下，你还是可以知道一个序列是否为无穷序列。如果是无穷序列，那么你自己可以写一个退出条件，以确保程序不会因为陷入无穷循环而崩溃。虽然 `for..of` 一般不会遇到这个问题，但如果遇到了却又没有退出条件，在使用扩展操作符或 `Array.from` 的情况下，程序就会迅速崩溃。

介绍完创建可迭代对象的技术细节后，接下来看看如何在实践中使用迭代器。

4.2.3 迭代对象以生成键/值对

将普通对象转换为可迭代对象有非常多的实用场景。对象映射、希望可以迭代的伪数组、4.2.2 节中的随机数生成器，以及属性通常可以迭代的类和普通对象，都可以通过遵循可迭代协议而获益。

通常来说，我们会用 JavaScript 对象来表示字符串键与任意值之间的映射。以下代码创建了一个颜色名称与相应的十六进制 RGB 值之间的映射。有时可能需要遍历其中的颜色名、十六进制值或键 / 值对。

```
const colors = {
  green: '#0e0',
  orange: '#f50',
  pink: '#e07'
}
```

以下代码为 `colors` 映射实现了产生 `[key, value]` 序列的可迭代能力。因为遵循迭代器协议给 `Symbol.iterator` 属性赋值了，所以遍历这个映射的键和值就变得非常简单了。

```
const colors = {
  green: '#0e0',
  orange: '#f50',
  pink: '#e07',
```

```
  [Symbol.iterator]() {
    const keys = Object.keys(colors)
    return {
      next() {
        const done = keys.length === 0
        const key = keys.shift()
        return {
          done,
          value: [key, colors[key]]
        }
      }
    }
  }
}
```

如果想要取得所有的键 / 值对，可以使用扩展操作符 `...`，如下所示。

```
console.log([...colors])
// <- [['green', '#0e0'], ['orange', '#f50'], ['pink', '#e07']]
```

但是，用一堆实现迭代器协议的代码“污染”本来很简单的 `colors` 映射是有问题的，因为可迭代能力本身与存储在这个映射中的颜色名称和十六进制值基本没什么关系。最好的办法是将添加键 / 值对迭代器的逻辑提取到一个可重用的函数里，从而使其与 `colors` 映射解耦。这样一来，我们可以将 `keyValueIterable` 函数单独放在某个地方，留作以后重用。

```
function keyValueIterable(target) {
  target[Symbol.iterator] = function () {
    const keys = Object.keys(target)
    return {
      next() {
        const done = keys.length === 0
        const key = keys.shift()
        return {
          done,
          value: [key, target[key]]
        }
      }
    }
  }
  return target
}
```

现在我们可以调用 `keyValueIterable` 并传入 `colors` 对象，从而将 `colors` 转换成一个可迭代对象。实际上，我们可以对任何想要迭代得到其键 / 值对的对象调用 `keyValueIterable`。迭代行为本身与对象保存的是什么信息无关。添加 `Symbol.iterator` 行为后，就可以将对象看作可迭代对象了。以下代码迭代了 `colors` 对象的键 / 值对，但只打印颜色值。

```
const colors = keyValueIterable({
  green: '#0e0',
  orange: '#f50',
```

```
  pink: '#e07'
})
for (const [ , color] of colors) {
  console.log(color)
  // <- '#0e0'
  // <- '#f50'
  // <- '#e07'
}
```

音乐播放器就是一个有趣的用例。

4.2.4 打造多功能播放列表

假设你正在开发一款音乐播放器，该播放器可以在播放完列表中的所有音乐之后就停止，也可以打开循环模式反复播放。只要有无穷循环列表的需求，就可以考虑使用可迭代协议。

如果有人在自己的资料库中添加了几首歌，你可以将它们保存在数组中，如下所示。

```
const songs = [
  'Bad moon rising – Creedence',
  'Don't stop me now – Queen',
  'The Scientist – Coldplay',
  'Somewhere only we know – Keane'
]
```

我们可以创建一个 `playlist` 函数，以返回一个表示当前应用要播放的曲目的序列。这个函数接收两个参数：`songs` 和 `repeat`，前者由用户提供，后者表示循环次数，如 1 次、2 次或无限循环（`Infinity`）。

播放列表 `playlist` 函数的实现如下所示。我们用从 0 开始的 `index` 值跟踪曲目序列中的音乐的位置，播放下一首曲目时就给 `index` 加 1，直到当前循环播放结束。此时，我们给 `repeat` 减 1 并将 `index` 重置为 0。没有可播放曲目且 `repeat` 为 0 时，播放结束。

```
function playlist(songs, repeat) {
  return {
    [Symbol.iterator]() {
      let index = 0
      return {
        next() {
          if (index >= songs.length) {
            repeat--
            index = 0
          }
          if (repeat < 1) {
            return { done: true }
          }
          const song = songs[index]
          index++
          return { done: false, value: song }
```

```
        }
      }
    }
  }
}
```

以下代码展示了 playlist 函数如何接收一个歌曲数组，并根据指定循环次数返回一个播放序列。如果循环次数为 Infinity，那么返回的序列也将是无穷序列，否则就是有限序列。

```
console.log([...playlist(['a', 'b'], 3)])
// <- ['a', 'b', 'a', 'b', 'a', 'b']
```

为了迭代播放列表，我们还需要编写一个 player 函数。假设已经有一个 playSong 函数可用于播放歌曲，并在播放结束后调用回调，那这个 player 函数可以按照如下方式来编写。总的来说，我们会异步循环传入序列的迭代器，在前一首歌曲播放结束后回调播放下一首。因为两次调用 g.next 会间隔一段时间，每首歌曲总需要一段时间才能播放完（播放歌曲是在 playSong 中实现的），所以因无限循环导致程序卡死的风险很小。即使 playlist 函数返回的是无穷序列，问题也不大。

```
function player(sequence) {
  const g = sequence[Symbol.iterator]()
  more()
  function more() {
    const item = g.next()
    if (item.done) {
      return
    }
    playSong(item.value, more)
  }
}
```

所有功能齐备之后，重复播放一个音乐列表只需要几行代码，如下所示。

```
const songs = [
  'Bad moon rising - Creedence',
  'Don't stop me now - Queen',
  'The Scientist - Coldplay',
  'Somewhere only we know - Keane'
]
const sequence = playlist(songs, Infinity)
player(sequence)
```

此时要加入随机播放功能也不难。只要稍微修改一下 playlist 函数，添加一个 shuffle 标签，如果用户传入了这个参数，对播放列表随机重新排序即可。

```
function playlist(inputSongs, repeat, shuffle) {
  const songs = shuffle ? shuffleSongs(inputSongs) : inputSongs
  return {
    [Symbol.iterator]() {
```

```
        let index = 0
        return {
          next() {
            if (index >= songs.length) {
              repeat--
              index = 0
            }
            if (repeat < 1) {
              return { done: true }
            }
            const song = songs[index]
            index++
            return { done: false, value: song }
          }
        }
      }
    }
}
function shuffleSongs(songs) {
  return songs.slice().sort(() => Math.random() > 0.5 ? 1 : -1)
}
```

如果想要打乱播放次序，那么必须将 `shuffle` 参数设置为 `true`。否则，歌曲还是会按照用户添加的顺序播放。这里又一次将列表排序的逻辑抽象了出来，与前面的函数实现了解耦。`playlist` 函数仍然只专注于生成供播放器播放的曲目序列。

```
console.log([...playlist(['a', 'b'], 3, true)])
// <- ['a', 'b', 'b', 'a', 'a', 'b']
```

你可能会说，`playlist` 其实也不该管排序的事。更好的设计思路可能是将重排函数放入调用代码。如果 `playlist` 还是原样，没有 `shuffle` 参数，我们仍然可以通过以下代码生成重排后的曲目序列。

```
function shuffleSongs(songs) {
  return songs.slice().sort(() => Math.random() > 0.5 ? 1 : -1)
}
console.log([...playlist(shuffleSongs(['a', 'b']), 3)])
// <- ['a', 'b', 'b', 'a', 'a', 'b']
```

作为 ES6 中的一个重要工具，迭代器不仅能帮助我们解耦代码，还能帮助我们实现之前很难实现的构造，如模糊处理曲目序列的能力，无论序列是无穷的还是有限的。从某种程度上来说，这种无差别处理令使用迭代器协议的代码写起来更优雅。当然，将未知的可迭代对象转换成数组时（如通过扩展操作符）也有一定的风险。如果碰巧要转换的是一个无穷序列，那可能会导致程序崩溃。

生成器也是一种返回可迭代对象的函数，但使用它不必显式声明带有 `Symbol.iterator` 方法的对象字面量。正因为如此，用生成器实现 4.2.2 节中的 `range` 和 `take` 函数更容易，而且它还有一些更有意思的用例。

4.3 生成器函数与生成器对象

生成器是 ES6 的新特性。生成器的定义方式是先创建一个函数，调用这个函数再返回生成器对象 g。这个 g 是一个可迭代对象，我们可以通过 Array.from(g)、[...g] 或 for..of 循环来使用它。生成器函数允许我们声明一种特殊的迭代器。这种迭代器会推迟代码的执行，同时保持自己的上下文。

4.3.1 生成器基础

上一节介绍过迭代器，我们知道每次迭代都会调用 .next() 方法从序列中取一个值。在生成器中看不到返回值的 next 方法，只能看到向序列中添加值的 yield 关键字。

以下是一个生成器函数的示例。注意 function 关键字后面的星号 *。这并不是我们打错了，星号是生成器函数的标志。

```
function* abc() {
  yield 'a'
  yield 'b'
  yield 'c'
}
```

生成器对象同时遵守可迭代协议和迭代器协议。

- 生成器对象 chars 是通过函数 abc 创建的。
- 对象 chars 是一个可迭代对象，因为它有一个 Symbol.iterator 方法。
- 对象 chars 也是一个迭代器，因为它有一个 .next 方法。
- chars 的迭代器就是它自己。

以上表述用 JavaScript 代码来证实，如下所示。

```
const chars = abc()
typeof chars[Symbol.iterator] === 'function'
typeof chars.next === 'function'
chars[Symbol.iterator]() === chars
console.log(Array.from(chars))
// <- ['a', 'b', 'c']
console.log([...chars])
// <- ['a', 'b', 'c']
```

创建生成器对象时，我们会得到一个用生成器函数产生可迭代序列的迭代器。每当代码执行到 yield 表达式，迭代器就会返回该表达式的值，而且生成器函数会暂停执行。

以下示例表明迭代会触发生成器函数中的副作用。当生成器函数恢复执行以返回序列中下一个元素时，每个 yield 语句后面的 console.log 语句都会执行。

```
function* numbers() {
  yield 1
  console.log('a')
  yield 2
  console.log('b')
  yield 3
  console.log('c')
}
```

假设你创建了生成器函数 numbers，然后通过扩展操作符将其内容保存为一个数组，并打印到控制台。思考 numbers 函数中的副作用，你觉得以下代码执行后控制台会输出什么？扩展操作符会迭代整个序列，以帮助你创建一个数组，在以下的 console.log 语句打印出数组之前，所有的副作用都会在通过解构构建数组的过程中被执行。

```
console.log([...numbers()])
// <- 'a'
// <- 'b'
// <- 'c'
// <- [1, 2, 3]
```

如果换作使用 for..of 循环，那就可以保持 numbers 函数中声明的顺序了。以下示例用 for..of 循环逐个打印了 numbers 序列中的每个元素。第一次从生成器函数中取 number 时，它会返回 1 并暂停执行。第二次，生成器函数会从上次暂停的地方恢复并打印出副作用 a，接着返回 2。第三次的副作用是 b，并返回了 3。第四次的副作用是 c，此时生成器指示序列已结束。

```
for (const number of numbers()) {
  console.log(number)
  // <- 1
  // <- 'a'
  // <- 2
  // <- 'b'
  // <- 3
  // <- 'c'
}
```

使用 yield* 委托生成序列

生成器函数可以使用 yield* 将生成序列的任务委托给一个生成器对象或其他可迭代对象。

因为 ES6 中的字符串遵守可迭代协议，所以可以通过以下代码将字符串 hello 拆分为字母。

```
function* salute() {
  yield* 'hello'
}
console.log([...salute()])
// <- ['h', 'e', 'l', 'l', 'o']
```

当然，此时直接使用 [...'hello'] 更简单。然而，有多条 yield 语句时，委托的作用就体现出来了。以下的示例修改了 salute 生成器来接收一个参数 name，从而生成了包含字符串 'hello you' 中所有字符的数组。

```
function* salute(name) {
  yield* 'hello '
  yield* name
}
console.log([...salute('you')])
// <- ['h', 'e', 'l', 'l', 'o', ' ', 'y', 'o', 'u']
```

再次强调，我们可以通过 yield* 将生成序列的任务委托给任何遵守可迭代协议的对象，而不仅仅是字符串。这些对象包括生成器对象、数组、arguments、浏览器中的 NodeList，总之只要实现了 Symbol.iterator 就可以。以下示例展示了如何同时使用 yield 和 yield*，并组合其他生成器函数、可迭代对象和扩展操作符来描述一个值序列。你能推断出 console.log 语句会打印出什么结果吗？

```
const salute = {
  [Symbol.iterator]() {
    const items = ['h', 'e', 'l', 'l', 'o']
    return {
      next: () => ({
        done: items.length === 0,
        value: items.shift()
      })
    }
  }
}
function* multiplied(base, multiplier) {
  yield base + 1 * multiplier
  yield base + 2 * multiplier
}
function* trailmix() {
  yield* salute
  yield 0
  yield* [1, 2]
  yield* [...multiplied(3, 2)]
  yield [...multiplied(6, 3)]
  yield* multiplied(15, 5)
}
console.log([...trailmix()])
```

以下就是生成器函数 trailmix 返回的序列。

```
['h', 'e', 'l', 'l', 'o', 0, 1, 2, 5, 7, [9, 12], 20, 25]
```

除了使用扩展操作符、for..of 和 Array.from 来迭代生成器对象，我们还可以直接手工迭代生成器对象。下面就来看看怎么做。

4.3.2 手工迭代生成器

生成器迭代并不限于 for..of、Array.from 或扩展操作符。与任何可迭代对象一样，有 Symbol.iterator 就可以通过 .next 按需取值，而不必像使用 for..of 那样严格同步，或像使用 Array.from 和扩展操作符那样一次性取出所有值。因为生成器对象既是可迭代对象也是迭代器，所以不必调用 g[Symbol.iterator]() 返回迭代器，g 本身就是迭代器，它与调用 Symbol.iterator 方法返回的对象是同一个对象。

假设有前面创建的迭代器 numbers，以下示例展示了如何使用生成器对象和 while 循环来手工迭代它。记住，迭代器返回的元素都有一个表示序列是否结束的 done 属性以及一个表示序列中当前值的 value 属性。

```
const g = numbers()
while (true) {
  const item = g.next()
  if (item.done) {
    break
  }
  console.log(item.value)
}
```

与 for..of 相比，用迭代器遍历生成器看起来有点复杂，但它适合一些有意思的用例。特别是 for..of 始终是一个同步循环，而有了迭代器的话，何时调用 g.next 就由我们说了算。此外，这个优势还衍生出了更多机会，比如，我们可以在获取某个异步操作的结果后再调用 g.next。

每次在生成器上调用 .next() 都可能有四种“事件”导致生成器内部暂停执行，并给 .next() 的调用者返回一个结果。简单总结如下。

- yield 表达式返回序列中的下一个值。
- return 语句返回序列中的最后一个值。
- throw 语句完全中断生成器的执行。
- 到达生成器函数的最后，获取值 { done: true }，因为函数隐式地返回 undefined。

迭代完生成器 g 的整个序列后，再调用 g.next() 不会有任何变化，只会返回 { done: true }。以下代码表明，到达序列的最后时，每次调用 g.next 都会返回相同的结果。

```
function* generator() {
  yield 'only'
}
const g = generator()
console.log(g.next())
// <- { done: false, value: 'only' }
console.log(g.next())
// <- { done: true }
console.log(g.next())
// <- { done: true }
```

4.3.3 将生成器混入可迭代对象

我们来快速回顾一下生成器。生成器函数被调用时会返回生成器对象。生成器对象有一个 `next` 方法，用于返回序列中的下一个元素。`next` 方法返回的对象形式为 `{ value, done }`。

以下示例定义了一个生成无穷斐波纳契序列的生成器。然后我们实例化了生成器对象并读取序列中的前 8 个值。

```
function* fibonacci() {
  let previous = 0
  let current = 1
  while (true) {
    yield current
    const next = current + previous
    previous = current
    current = next
  }
}
const g = fibonacci()
console.log(g.next()) // <- { value: 1, done: false }
console.log(g.next()) // <- { value: 1, done: false }
console.log(g.next()) // <- { value: 2, done: false }
console.log(g.next()) // <- { value: 3, done: false }
console.log(g.next()) // <- { value: 5, done: false }
console.log(g.next()) // <- { value: 8, done: false }
console.log(g.next()) // <- { value: 13, done: false }
console.log(g.next()) // <- { value: 21, done: false }
```

用可迭代对象来实现与此类似。只不过它要遵守协议，返回的对象必须有一个 `next` 方法。这个方法应该返回形如 `{ value, done }` 的序列元素。以下示例展示了一个可迭代对象 `fibonacci`，它与前面生成器的功能大致相同。

```
const fibonacci = {
  [Symbol.iterator]() {
    let previous = 0
    let current = 1
    return {
      next() {
        const value = current
        const next = current + previous
        previous = current
        current = next
        return { value, done: false }
      }
    }
  }
}
const sequence = fibonacci[Symbol.iterator]()
console.log(sequence.next()) // <- { value: 1, done: false }
console.log(sequence.next()) // <- { value: 1, done: false }
console.log(sequence.next()) // <- { value: 2, done: false }
```

```
console.log(sequence.next()) // <- { value: 3, done: false }
console.log(sequence.next()) // <- { value: 5, done: false }
console.log(sequence.next()) // <- { value: 8, done: false }
console.log(sequence.next()) // <- { value: 13, done: false }
console.log(sequence.next()) // <- { value: 21, done: false }
```

再次强调，可迭代对象会返回一个有 next 方法的对象，生成器函数也会这么做。next 方法应该返回一个形如 { value, done } 的对象，生成器函数也一样。如果我们修改可迭代对象 fibonacci，让它的 Symbol.iterator 属性指向一个生成器函数，会怎么样呢？事实证明，它照样会工作。

以下示例表明，可迭代对象 fibonacci 使用一个生成器函数来定义自己如何被迭代。注意，这个可迭代对象与前面的 fibonacci 生成器函数几乎完全一样。不仅如此，我们仍然可以使用生成器函数中使用的所有语义，如 yield、yield*。

```
const fibonacci = {
  * [Symbol.iterator]() {
    let previous = 0
    let current = 1
    while (true) {
      yield current
      const next = current + previous
      previous = current
      current = next
    }
  }
}
const g = fibonacci[Symbol.iterator]()
console.log(g.next()) // <- { value: 1, done: false }
console.log(g.next()) // <- { value: 1, done: false }
console.log(g.next()) // <- { value: 2, done: false }
console.log(g.next()) // <- { value: 3, done: false }
console.log(g.next()) // <- { value: 5, done: false }
console.log(g.next()) // <- { value: 8, done: false }
console.log(g.next()) // <- { value: 13, done: false }
console.log(g.next()) // <- { value: 21, done: false }
```

此外，可迭代协议同样有效。为了验证这一点，我们可以使用 for..of 这样的构造函数，而非手工创建的生成器对象。为了防止无穷序列造成程序崩溃，以下示例中的 for..of 循环添加了“断路保护”装置。

```
for (const value of fibonacci) {
  console.log(value)
  if (value > 20) {
    break
  }
}
// <- 1
// <- 1
// <- 2
```

```
// <- 3
// <- 5
// <- 8
// <- 13
// <- 21
```

接下来我们再看几个更接地气的示例，看看在迭代树形数据结构时使用生成器代码会有多简洁。

4.3.4 使用生成器遍历树

基于树形数据结构的算法并不好理解，因为其中经常涉及递归。以下代码定义了一个 `Node` 类，它有一个 `value` 以及任意个数的子节点。

```
class Node {
  constructor(value, ...children) {
    this.value = value
    this.children = children
  }
}
```

我们可以使用深度优先的搜索算法来遍历树，也就是优先进入树形结构中更深的层次，直到没有子节点为止。在以下的树形结构中，深度优先的搜索算法会按照从 `1~10` 的顺序造访每个节点。

```
const root = new Node(1,
  new Node(2),
  new Node(3,
    new Node(4,
      new Node(5,
        new Node(6)
      ),
      new Node(7)
    )
  ),
  new Node(8,
    new Node(9),
    new Node(10)
  )
)
```

要想实现深度优先的遍历算法，可以定义一个生成器函数，使其返回当前节点的值，然后再迭代其子节点。具体就是用 `yield*` 操作符来拼接迭代器递归的结果，返回序列中的每一项。

```
function* depthFirst(node) {
  yield node.value
  for (const child of node.children) {
    yield* depthFirst(child)
```

```
  }
}
console.log([...depthFirst(root)])
// <- [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

另一种方式是使用上面的 depthFirst 生成器作为迭代器，将 Node 转换成可迭代对象（类）。以下代码也利用了 child 是 Node 这一点，即 child 也是一个可迭代对象的事实。因此才能用 yield* 返回作为父节点元素的 child 的可迭代序列。

```
class Node {
  constructor(value, ...children) {
    this.value = value
    this.children = children
  }
  * [Symbol.iterator]() {
    yield this.value
    for (const child of this.children) {
      yield* child
    }
  }
}
console.log([...root])
// <- [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

如果想要使用宽度优先的算法，那么可以将迭代器修改成以下代码那样。这里使用了“先进先出”队列来保存尚未访问的节点。迭代的每一步都从 root 节点开始，我们会打印当前节点的值，并将其所有子元素推到队列中。子元素始终会追加到队尾，但我们是从队列开头取元素的。这意味着我们会先将某一层次的节点都遍历完，然后再进入下一层。

```
class Node {
  constructor(value, ...children) {
    this.value = value
    this.children = children
  }
  * [Symbol.iterator]() {
    const queue = [this]
    while (queue.length) {
      const node = queue.shift()
      yield node.value
      queue.push(...node.children)
    }
  }
}
console.log([...root])
// <- [1, 2, 3, 8, 4, 9, 10, 5, 7, 6]
```

极强的表达力使得生成器非常有用，但我们可以用迭代器协议来定义一个按照自己的想法迭代的序列。在碰到某个树形结构包含几千个节点，且出于性能考虑需要节流迭代操作时，生成器是非常方便的。

4.3.5 传递生成器函数

到目前为止，我们的讨论一直围绕如何通过生成器构建方便使用的序列进行。生成器也可以作为一段代码的接口，至于生成器函数如何被迭代，则由这段代码控制。

我们将在本节编写一个生成器函数，并直接将它传递给一个方法，这个方法会遍历生成器，并使用其序列中的元素。乍一看你可能会觉得这样编写代码有点不合常规。事实上，很多基于生成器的库都是由用户提供生成器的，库本身只管控制迭代。

以下代码展示了我们希望见到的 `modelProvider` 方法的使用方式。也就是说，我们希望这个方法的用户自己提供一个生成器函数，该函数要 `yield` 一个模型不同属性的访问路径，之后再取得该模型中指定路径的结果。生成器对象可以通过 `g.next(result)` 将结果传回生成器函数。传回生成器函数后，对 `yield` 表达式求值的结果就是生成器对象传回的 `result`。

```
modelProvider(function* () {
  const items = yield 'cart.items'
  const item = items.reduce(
    (left, right) => left.price > right.price ? left : right
  )
  const details = yield `products.${ item.id }`
  console.log(details)
})
```

用户提供的生成器每次 `yield` 一个路径后，生成器函数就会暂停执行，直到迭代器再次调用 `g.next()`，这个过程甚至可以在后台异步进行。以下是 `modelProvider` 方法的实现，其代码负责迭代生成器返回的路径序列。注意，`data` 通过 `g.next(data)` 又传回了生成器函数。

```
const model = {
  cart: {
    items: [item1, ..., itemN]
  },
  products: {
    product1: { ... },
    productN: { ... }
  }
}
function modelProvider(paths) {
  const g = paths()
  pull()
  function pull(data) {
    const { value, done } = g.next(data)
    if (done) {
      return
    }
    const crumbs = value.split('.')
    const data = crumbs.reduce(followCrumbs, model)
    pull(data)
  }
}
function followCrumbs(data, crumb) {
```

```
    if (!data || !data.hasOwnProperty(crumb)) {
      return null
    }
    return data[crumb]
  }
```

让用户提供生成器函数的最大好处是，通过为他们提供 yield 关键字产生了一种可能性：如果迭代器在两次 g.next() 调用之间执行异步操作，那么它们的代码是可以暂停的。下一节将探讨生成器在编写异步代码时的用处。

4.3.6　处理异步流

回到前面向 modelProvider 传入用户提供的生成器的示例。如果需要异步请求获取模型，那应该如何修改代码呢？先说生成器这边，如果迭代路径序列的方式变成异步，用户提供的函数也不用进行修改。生成器已经支持在查询模型时暂停了，这正是生成器的优点的体现。需要修改的只是向某个服务请求相关路径的查询结果，再通过某种方式返回这个结果，然后在生成器对象上调用 g.next 以通过居间的 yield 语句将结果传回生成器函数。

假设以下 modelProvider 的使用方式不变。

```
modelProvider(function* () {
  const items = yield 'cart.items'
  const item = items.reduce(
    (left, right) => left.price > right.price ? left : right
  )
  const details = yield `products.${ item.id }`
  console.log(details)
})
```

我们要用 fetch 请求 HTTP 资源，它返回的是一个 Promise。注意，我们不能在异步执行流中使用 for..of 来遍历序列，它只能用于同步模式。

以下代码会将请求通过 HTTP 发送到模型，现在由服务器负责查询交付结果。除了保存相关的用户认证数据（如 cookie），客户端不用再记录状态了。

```
function modelProvider(paths) {
  const g = paths()
  pull()
  function pull(data) {
    const { value, done } = g.next(data)
    if (done) {
      return
    }
    fetch(`/model?query=${ encodeURIComponent(value) }`)
      .then(response => response.json())
      .then(data => pull(data))
  }
}
```

务必记住，在对 yield 表达式求值时，生成器函数会暂停执行，直到迭代器请求序列中的下一项（这个示例中就是要查询的模型的下一个路径）时才会恢复执行。就此而言，即使 yield 会在下一次调用 g.next 之前暂停函数执行，生成器函数中的代码看起来也像是同步的。

虽然生成器可以让我们写出像同步代码那样的异步代码，但仍然存在问题。如果迭代过程中出错，如何处理错误呢？比如，要是 HTTP 请求失败，如何通知生成器，然后在生成器函数中处理错误呢？

4.3.7 在生成器上抛出错误

我们知道生成器有一个 g.throw 方法，用于在暂停期间报告错误。但如果不分析用户提供的生成器（此时由于 yield 导致暂停，它还保持着对看似同步的函数的控制），可能很难找到应该在哪里调用 g.throw。当我们将目光转移到 yield 表达式之间的流控制代码（正是这里可能出错）时，在哪里调用 g.throw 就显而易见了。如果处理序列中的某一项时发生错误，那么使用生成器的代码应该将错误抛到生成器中。

对于 modelProvider 来说，迭代器可能会遭遇网络问题（或格式不正确的 HTTP 响应），因而无法拿到模型的查询结果。以下代码给 fetch 添加了一个错误回调，它会在 response.json() 出错时执行，此时我们会在生成器上抛出异常。

```
fetch(`/model?query=${ encodeURIComponent(value) }`)
  .then(response => response.json())
  .then(data => pull(data))
  .catch(err => g.throw(err))
```

调用 g.next 时，生成器函数的代码会恢复执行。调用 g.throw 也会让生成器函数恢复执行，只不过恢复执行后会导致 yield 表达式抛出异常。生成器中未处理的异常会导致迭代终止，因为根本无法到达其他 yield 表达式。生成器函数可以将 yield 表达式包装在 try/catch 块中，从而恰当地处理迭代代码转发过来的异常，如下所示。这样后面的 yield 表达式也会正常地暂停生成器执行，将控制权再次交给迭代代码。

```
modelProvider(function* () {
  try {
    console.log('items in the cart:', yield 'cart.items')
  } catch (e) {
    console.error('uh oh, failed to fetch model.cart.items!', e)
  }
  try {
    console.log(`these are our products: ${ yield 'products' }`)
  } catch (e) {
    console.error('uh oh, failed to fetch model.products!', e)
  }
})
```

虽然可以通过生成器函数暂停执行，然后异步恢复，但我们仍然可以像在常规函数中那样

使用相同的错误处理方式（try、catch 和 throw）。在生成器函数中使用 try/catch 块，让我们能够像处理同步代码那样处理异步错误，即使迭代器代码中的 yield 表达式背后有 HTTP 请求。

4.3.8　代替生成器返回

除了 g.next 和 g.throw，生成器对象还有一个方法可以用来决定生成器序列如何被迭代：g.retrun(value)。这个方法会恢复生成器函数的执行，并在原来 yield 语句的位置执行 return value，通常这会终止对生成器序列的迭代。换句话说，这与生成器函数中有一条 return 语句效果相同。

```
function* numbers() {
  yield 1
  yield 2
  yield 3
}
const g = numbers()
console.log(g.next())
// <- { done: false, value: 1 }
console.log(g.return())
// <- { done: true }
console.log(g.next())
// <- { done: true }
```

考虑到 g.return(value) 会在生成器函数暂停时到达的 yield 的位置执行 return value，使用 try/finally 块可以避免立即终止迭代序列，因为 finally 块内的代码会在执行流退出函数前执行。这意味着 finally 块中的 yield 表达式会继续回送序列中的值，如下所示。

```
function* numbers() {
  try {
    yield 1
  } finally {
    yield 2
    yield 3
  }
  yield 4
  yield 5
}
const g = numbers()
console.log(g.next())
// <- { done: false, value: 1 }
console.log(g.return(-1))
// <- { done: false, value: 2 }
console.log(g.next())
// <- { done: false, value: 3 }
console.log(g.next())
// <- { done: true, value -1 }
```

接下来我们再看一个简单的生成器函数，它会先回送几个值，然后执行 return 语句。

```
function* numbers() {
  yield 1
  yield 2
  return 3
  yield 4
}
```

如果使用扩展操作符、Array.from 或 for..of 来迭代这个生成器，那么无论 return 语句放在什么位置，结果中都不会包含返回的 value，如下所示。

```
console.log([...numbers()])
// <- [1, 2]
console.log(Array.from(numbers()))
// <- [1, 2]
for (const number of numbers()) {
  console.log(number)
  // <- 1
  // <- 2
}
```

之所以会这样，是因为 g.return 或 return 语句返回给迭代器的结果中包含 done: true 这个标志，以表明该序列已经结束。就算返回的结果中确实也包含一个序列的 value，但上述方法都会忽略它。由此看来，对生成器而言，return 语句应该主要用于提供“断路保护”功能，而不是提供序列中的最后一个值。

取得生成器返回值的唯一方法是，迭代生成器对象时将结果包含 done: true 的情况考虑在内，如下所示。

```
const g = numbers()
console.log(g.next())
// <- { done: false, value: 1 }
console.log(g.next())
// <- { done: false, value: 2 }
console.log(g.next())
// <- { done: true, value: 3 }
console.log(g.next())
// <- { done: true }
```

由于 yield 表达式和 return 语句很容易混淆，最好不要在生成器中使用 return，除非有方法需要故意利用 yield 和 return 的区别。当然，终极目标还是要提取一层抽象，以简化开发的工作量。

下一节将利用 yield 和 return 的区别来创建一个迭代器，然后基于相同的生成器函数实现输入和输出。

4.3.9 基于生成器的异步I/O

以下代码展示了一个简单的生成器函数，其作用一目了然，前两行表示输入来源，第三行表示输出目标。这个假想的方法可用于从回送（yield）端取得产品信息，最终将这些信息

保存到返回（return）端。对于这个接口而言，用户不必花时间思考如何读写信息（这一点非常重要），只要提供来源和目标，剩下的由底层实现负责就行了。

```
saveProducts(function* () {
  yield '/products/modern-javascript'
  yield '/products/mastering-modular-javascript'
  return '/wishlists/books'
})
```

不仅如此，saveProducts 方法还可以返回一个 Promise，该 Promise 会在订单被推送到返回端后得到解决。换句话说，用户可以在订单被记录后执行回调。而且，生成器函数也应该通过 yield 表达式接收商品数据，这些数据可以通过调用 g.next 传进来。

```
saveProducts(function* () {
  const p2 = yield '/products/modern-javascript'
  const p2 = yield '/products/mastering-modular-javascript'
  return '/wishlists/books'
}).then(response => {
  // 保存商品列表后继续执行其他操作
})
```

通过引入条件逻辑，可以将 saveProducts 的目标设为用户的购物车，而不是心愿列表。

```
saveProducts(function* () {
  yield '/products/modern-javascript'
  yield '/products/mastering-modular-javascript'
  if (addToCart) {
    return '/cart'
  }
  return '/wishlists/books'
})
```

之所以采用这种“输入和输出”全包的方式，是因为这样即便实现有各种各样的变化，API 也可以基本保持不变。输入资源可以通过 HTTP 请求或从某个临时缓存取得，可以一个个获取（或同时获取），或者通过一种机制将所有回送的资源组合为一个 HTTP 请求。一次取一个与组合为一个请求在语义上有区别，但在实现变化如此之大的情况下，API 也不用变化。

接下来我们会一步步实现 saveProducts。首先，以下代码展示了如何通过 fetch 及其基于 Promise 的 API 发送 HTTP 请求，以获取第一个回送商品的 JSON 信息。

```
function saveProducts(productList) {
  const g = productList()
  const item = g.next()
  fetch(item.value)
    .then(res => res.json())
    .then(product => {})
}
```

要想连续异步地取得每个商品的信息（每次只取一个），我们可以将 fetch 调用封装在一个递归函数中，以便在取得每个商品的信息后再次调用它。这样每一步取得一个商品，然后调用 g.next 来恢复生成器函数的执行，让它回送序列中的下一项，然后再用该项调用 more。

```
function saveProducts(productList) {
  const g = productList()
  more(g.next())
  function more(item) {
    if (item.done) {
      return
    }
    fetch(item.value)
      .then(res => res.json())
      .then(product => {
        more(g.next(product))
      })
  }
}
```

这样我们就能取得所有输入（每次一个），并通过 g.next(product) 将结果返回给生成器。为了利用 return 语句，我们可以将取得的商品信息保存在一个临时数组中，等迭代器取得序列被标记为结束的 item 时，再用 POST 方法将这个数组发送到 item 表示的输出目标上。

```
function saveProducts(productList) {
  const products = []
  const g = productList()
  more(g.next())
  function more(item) {
    if (item.done) {
      save(item.value)
    } else {
      details(item.value)
    }
  }
  function details(endpoint) {
    fetch(endpoint)
      .then(res => res.json())
      .then(product => {
        products.push(product)
        more(g.next(product))
      })
  }
  function save(endpoint) {
    fetch(endpoint, {
      method: 'POST',
      body: JSON.stringify({ products })
    })
  }
}
```

这一步取得的商品信息会缓存在 products 数组中，传回生成器函数，并最终保存到 return

语句返回的输出目标。

前面我们提到 saveProducts 可以返回一个 Promise，这样保存之后还可以继续执行回调。正如前面所说的，fetch 返回 Promise。只要给每个函数调用都补上 return 语句，就可以让 saveProducts 返回 more 的输出结果，而后者又会返回 save 或 details 的输出。save 和 details 都返回 fetch 调用创建的 Promise。此外，每个 details 调用都会在自己的 Promise 中返回调用 more 的结果。这意味着最初的 fetch 在第二个 fetch 兑现前不会被兑现，于是我们就实现了 Promise 链。最终，所有的 Promise 会在 save 被调用并解决后得到解决。

```
function saveProducts(productList) {
  const products = []
  const g = productList()
  return more(g.next())
  function more(item) {
    if (item.done) {
      return save(item.value)
    }
    return details(item.value)
  }
  function details(endpoint) {
    return fetch(endpoint)
      .then(res => res.json())
      .then(product => {
        products.push(product)
        return more(g.next(product))
      })
  }
  function save(endpoint) {
    return fetch(endpoint, {
        method: 'POST',
        body: JSON.stringify({ products })
      })
      .then(res => res.json())
  }
}
```

你可能已经注意到了，以上实现并没有硬编码任何重要的操作。也就是说，只要有零个或多个输入，而你希望将它们发送到一个输出，那么就可以使用这种通用的输入和输出模式。用户看到的则是一个简简单单的、一目了然的方法，他们只要 yield 输入来源，return 输出目标就行了。此外，使用 Promise 也让用户可以继续连缀更多操作。这样一来，我们就可以将容易搞错的条件逻辑和执行流的控制权掌握在自己手里，因为执行流已经抽象到 saveProducts 方法的迭代机制中了。

到目前为止，我们已经讨论了很多流控制机制：回调、事件、Promise、迭代器和生成器。接下来的两节将探讨 async/await、异步迭代器和异步生成器，它们都构筑在混用前面几种流控制机制的基础之上。

4.4 异步函数

Python 和 C# 等语言都已经有了 `async/await`。在 ES2017 中，JavaScript 也提出了可用于描述异步操作的原生语法。

接下来我们将快速比较一下 Promise、回调和生成器。随后再看看 JavaScript 中的异步函数，看看它如何确保代码的可读性。

4.4.1 各种异步代码

假设有如下代码。这里我们将一个 `fetch` 请求包装在一个 `getRandomArticle` 函数中。返回的 Promise 中是 JSON 格式的响应体。如果失败，则执行标准的失败逻辑。

```
function getRandomArticle() {
  return fetch('/articles/random', {
    headers: new Headers({
      Accept: 'application/json'
    })
  })
  .then(res => res.json())
}
```

以下代码展示了如何使用 `getRandomArticle` 函数。我们构造了一个 Promise，它取得表示文章的 JSON 对象，然后将其传递给一个异步视图渲染函数 `renderView`，该函数返回 HTML 页面。获取 HTML 页面后，我们用该 HTML 内容替换当前页面。为了避免静默错误，`catch` 子句会通过 `console.error` 打印出错原因。

```
getRandomArticle()
  .then(model => renderView(model))
  .then(html => setPageContents(html))
  .then(() => console.log('Successfully changed page!'))
  .catch(err => console.error(err))
```

连缀起来的 Promise 会导致调试困难，找出流程报错的根本原因是一个艰巨的挑战。Promise 代码属于写起来容易、读起来难的那种类型。因此，Promise 代码也会提高维护成本。

如果这里使用简单的 JavaScript 回调，那么代码中就会出现重复，如下所示。与此同时，我们也会掉进“回调地狱”：异步代码流中的每一步都缩进了一级，步数越多，代码就越难理解。

```
getRandomArticle((err, model) => {
  if (err) {
    return console.error(err)
  }
  renderView(model, (err, html) => {
```

```
    if (err) {
      return console.error(err)
    }
    setPageContents(html, err => {
      if (err) {
        return console.error(err)
      }
      console.log('Successfully changed page!')
    })
  })
})
```

当然，有些库可以辅助处理“回调地狱”以及重复性错误。比如，async 库利用了常规的回调，其中第一个参数也是错误信息。使用其 waterfall 方法可以使前面的代码变得更简洁。

```
async.waterfall([
  getRandomArticle,
  renderView,
  setPageContents
], (err, html) => {
  if (err) {
    return console.error(err)
  }
  console.log('Successfully changed page!')
})
```

下面再看一个类似的示例，但这次使用生成器。以下代码是基于 getRandomArticle 的逻辑改写的，唯一的目的就是换一种实现方式使用生成器。

```
function getRandomArticle(gen) {
  const g = gen()
  fetch('/articles/random', {
    headers: new Headers({
      Accept: 'application/json'
    })
  })
  .then(res => res.json())
  .then(json => g.next(json))
  .catch(err => g.throw(err))
}
```

以下代码展示了如何通过一个 yield 表达式从 getRandomArticle 中获取 json。虽然代码看起来像是同步的，但其中封装了一个生成器。要想添加更多步骤，就得大幅修改 getRandomArticle，以便它可以返回我们想要的结果，同时还要对生成器函数进行必要的修改，以便接收更新后的结果。

```
getRandomArticle(function* printRandomArticle() {
  const json = yield
  // render view
})
```

在这种情况下，使用生成器也许并不是最直观的实现方式，而且这只是将复杂性转移了。相比之下，或许我们还是应该继续使用 Promise。

除了引入不直观的代码，迭代器代码也会与所使用的生成器函数高度耦合。这意味着，只要给生成器添加新的 yield 表达式，就得修改迭代器。

此时，异步函数是一个更好的选择。

4.4.2 使用async/await

async 函数兼顾了基于 Promise 的实现和生成器风格的同步写法。这个方法的巨大优势是，你根本不用修改原始的 getRandomArticle 函数，只要它返回 Promise，能加 await 就行。

注意，await 关键字只能用于标记为 async 的异步函数内部。async 函数的原理与生成器类似，只不过是在局部上下文中挂起，直至 Promise 返回。如果以 await 修饰的表达式不是 Promise，那它会被转换为 Promise。

以下代码使用了最初的 getRandomArticle 函数，仍然以 Promise 为基础。然后将 model 传递给另一个异步函数 renderView，这个函数又返回一小段 HTML，接着再更新页面。注意，在标记为 async 的函数中等待多个 Promise 逐个解决时，我们使用了 try/catch 来处理错误。这完全就是同步代码的写法，但那几个函数又确实是异步执行的。

```
async function read() {
  try {
    const model = await getRandomArticle()
    const html = await renderView(model)
    await setPageContents(html)
    console.log('Successfully changed page!')
  } catch (err) {
    console.error(err)
  }
}

read()
```

异步函数始终会返回 Promise。如果有未捕获的异常，那么返回的 Promise 的结果是被拒绝；否则，返回的 Promise 将以返回值被解决。利用异步函数的这一特点，我们可以将其与常规的 Promise 连缀调用结合起来。以下代码展示了二者的结合。

```
async function read() {
  const model = await getRandomArticle()
  const html = await renderView(model)
  await setPageContents(html)
  return 'Successfully changed page!'
}

read()
```

```
  .then(message => console.log(message))
  .catch(err => console.error(err))
```

为了让 `read` 函数更有用，我们可以让它返回结果 `html`，并允许使用者使用 Promise 或者另一个异步函数继续下去。这样一来，`read` 函数就只专注于为视图提供 HTML 了。

```
async function read() {
  const model = await getRandomArticle()
  const html = await renderView(model)
  return html
}
```

接下来我们可以通过普通的 Promise 直接打印 HTML。

```
read().then(html => console.log(html))
```

可见，使用异步函数实现连缀调用一点也不难。以下示例创建了一个 `write` 函数，用于通过调用 `read` 实现连缀。

```
async function write() {
  const html = await read()
  console.log(html)
}
```

那并发的异步流程怎么办呢？

4.4.3 并发异步流

在异步代码流中，并发执行两个甚至更多任务是常有的事。虽然异步函数可以简化异步代码的编写，但代码一次也只能执行一个异步操作。如果一个函数包含多个 `await` 表达式，那就要每次都挂起，直到 Promise 结束，再轮到下一个 `await` 表达式执行。这其实有点类似生成器中的 `yield`。

```
async function concurrent() {
const p1 = new Promise(resolve =>
setTimeout(resolve, 500, 'fast')
)
const p2 = new Promise(resolve =>
setTimeout(resolve, 200, 'faster')
)
const p3 = new Promise(resolve =>
setTimeout(resolve, 100, 'fastest')
)
const r1 = await p1 // 在p1返回之前，执行流是阻塞的
const r2 = await p2
const r3 = await p3
}
```

我们可以使用 `Promise.all` 解决这个问题，只创建一个 Promise 搭配一个 `await`。这样一

来，代码会一直阻塞到列表中的所有 Promise 都解决，也就是它们可以被并发地解决。

以下示例展示了如何用一个 await 挂起三个可以并发解决的 Promise。由于 await 挂起了 async 函数，表达式 await Promise.all 最终会解决为一个结果数组。我们可以用解构分别取出这个数组中的每个结果。

```
async function concurrent() {
const p1 = new Promise(resolve =>
setTimeout(resolve, 500, 'fast')
)
const p2 = new Promise(resolve =>
setTimeout(resolve, 200, 'faster')
)
const p3 = new Promise(resolve =>
setTimeout(resolve, 100, 'fastest')
)
const [r1, r2, r3] = await Promise.all([p1, p2, p3])
console.log(r1, r2, r3)
// 'fast', 'faster', 'fastest'
}
```

对于并发任务，我们还可以用 Promise.race 取得兑现最快的 Promise 返回的结果。

```
async function race() {
  const p1 = new Promise(resolve => setTimeout(resolve, 500, 'fast'))
  const p2 = new Promise(resolve => setTimeout(resolve, 200, 'faster'))
  const p3 = new Promise(resolve => setTimeout(resolve, 100, 'fastest'))
  const result = await Promise.race([p1, p2, p3])
  console.log(result)
  // 'fastest'
}
```

4.4.4　错误处理

与在普通的 Promise 中一样，async 函数中的错误也会被静默地吞掉，因为异步函数本身被包装在一个 Promise 中。异步函数体内或 await 挂起的异步任务抛出的未捕获错误，会导致异步函数返回的 Promise 被拒绝。

如果为 await 表达式包上一层 try/catch，那么对这部分被包装起来的异步函数代码而言，错误处理是按照典型的 try/catch 语义进行的。

当然，这可以看作异步函数的一个优势。毕竟，你没有办法对异步回调使用 try/catch，但使用 Promise 时倒可以。从这个意义上来说，异步函数有点类似于生成器。通过将函数执行挂起，生成器将异步流转换成了同步代码，因此我们同样可以使用 try/catch 结构。

此外，我们还可以在异步函数外部给它返回的 Promise 添加一个 .catch 子句来捕获上述异常。虽然错误处理既可以使用 try/catch，也可以使用 pomise 的 .catch，但除非所有人都

非常了解异步函数是通过 Promise 封装的这个语义，以及 try/catch 能在这里使用的原因是什么，否则这很容易造成认知误区，并最终导致错误不被处理。

```
read()
  .then(html => console.log(html))
  .catch(err => console.error(err))
```

正如你看到的，捕获、处理（或转移）异常及输出错误信息的手段还是蛮多的。

4.4.5　深入理解异步函数

异步函数内部同时利用了生成器和 Promise。假设有以下异步函数：

```
async function example(a, b, c) {
  // 函数体省略
}
```

以下代码展示了如何将 example 函数声明转换为一个简单的函数，该函数返回以一个生成器为参数的 spawn 辅助函数。

```
function example(a, b, c) {
  return spawn(function* () {
    // 函数体省略
  })
}
```

在生成器函数中，我们可以假设 yield 在语法上等价于 await。

在 spawn 函数中，Promise 对象包装了将要单步执行（由用户代码组成的）生成器函数的代码，并按顺序将值转发给生成器代码（async 函数的函数体）。

以下代码可以帮助你理解 async/await 通过生成器迭代一连串 await 表达式的实现过程。序列中的每个表达式都被包装在一个 Promise 中，从而与下一个表达式连接起来。如果整串 Promise 结束或者其中一个 Promise 被拒绝，则底层的生成器函数就会返回相应的 Promise。

```
function spawn(generator) {
  // 将所有代码包装在一个Promise中
  return new Promise((resolve, reject) => {
    const g = generator()

    // 运行第一步
    step(() => g.next())

    function step(nextFn) {
      const next = runNext(nextFn)
      if (next.done) {
        // 成功结束，解决当前Promise
        resolve(next.value)
        return
      }
```

```
      // 未结束，连缀返回的Promise并运行下一步
      Promise
        .resolve(next.value)
        .then(
          value => step(() => g.next(value)),
          err => step(() => g.throw(err))
        )
    }

    function runNext(nextFn) {
      try {
        // 继续执行生成器代码
        return nextFn()
      } catch (err) {
        // 以失败告终，拒绝当前Promise
        reject(err)
      }
    }
  })
}
```

我们来看看以下的异步函数示例。为打印结果，这里也使用了基于 Promise 的连缀。我们通过这个示例来演示一遍实现流程。

```
async function exercise() {
  const r1 = await new Promise(resolve =>
  setTimeout(resolve, 500, 'slowest')
)
  const r2 = await new Promise(resolve =>
  setTimeout(resolve, 200, 'slow')
)
  return [r1, r2]
}

exercise().then(result => console.log(result))
// <- ['slowest', 'slow']
```

首先，我们将这个函数转换为基于前面 `spawn` 的逻辑。也就是将这个异步函数的代码体包装到一个生成器中，再将生成器传给 `spawn`，同时在生成器中用 `yield` 代替 `await`。

```
function exercise() {
  return spawn(function* () {
    const r1 = yield new Promise(resolve =>
    setTimeout(resolve, 500, 'slowest')
  )
    const r2 = yield new Promise(resolve =>
    setTimeout(resolve, 200, 'slow')
  )
    return [r1, r2]
  })
}

exercise().then(result => console.log(result))
// <- ['slowest', 'slow']
```

当带有生成器的 spawn 函数被调用时，它会立即创建一个生成器对象，并执行一次 step，如下所示。之后只要遇到（与异步函数中的 await 表达式等价的）yield 表达式，就会再调用这个 step 函数。

```
function spawn(generator) {
  //将所有代码包装在一个Promise中
  return new Promise((resolve, reject) => {
    const g = generator()

    // 运行第一步
    step(() => g.next())
    // ...
  })
}
```

执行 step 函数的第一步是（通过 runNext 函数）在 try/catch 块中调用 nextFn 函数。这会恢复生成器函数的执行。如果生成器函数此时报错，则进入 catch 子句，异步函数的底层 Promise 就会被拒绝，不再继续下一步，如下所示。

```
function step(nextFn) {
  const next = runNext(nextFx)
  // ...
}

function runNext(nextFn) {
  try {
    // 继续执行生成器代码
    return nextFn()
  } catch (err) {
    // 以失败告终，拒绝当前Promise
    reject(err)
  }
}
```

回到异步函数，以下表达式的代码会被求值。没有什么错误，异步函数的执行再次被挂起。

```
yield new Promise(resolve =>
  setTimeout(resolve, 500, 'slowest')
)
```

这个 yield 表达式返回的结果在 step 函数中由 next.value 接收，而 next.done 表明生成器序列是否结束。对当前的示例而言，我们在函数中接收 Promise，并准确控制迭代顺序。此时 next.done 是 false，因此我们不会在这一步解决包装异步函数的 Promise。这里将 next.value 封装到一个兑现的 Promise 中，以防没有收到（或收到的不是）Promise。

然后就是等待这个 Promise 被兑现或被拒绝。如果这个 Promise 成功兑现，则将得到的 value 传递给生成器，同时将生成器序列向前推进一步。如果这个 Promise 被拒绝，就会通

过 runNext 函数调用 g.throw 在生成器函数中抛出异常，这个异常进而导致在 runNext 函数中将包装整个异步函数的 Promise 拒绝掉。

```
function step(nextFn) {
  const next = runNext(nextFn)
  if (next.done) {
    // 成功结束，解决当前Promise
    resolve(next.value)
    return
  }
  // 未结束，连缀返回的Promise并运行下一步
    .resolve(next.value)
    .then(
      value => step(() => g.next(value)),
      err => step(() => g.throw(err))
    )
}
```

如果只是单纯调用 g.next()，那么只会触发生成器函数继续执行。像这样给它传递一个值 g.next(value) 就能让对应的 yield 表达式在求值后取得这个值（value）。此时我们说的值也就是最初的 yield 的 Promise（即 'slowest'）兑现后得到的值。

回到生成器函数，我们将 'slowest' 赋值给 r1。

```
const r1 = yield new Promise(resolve =>
  setTimeout(resolve, 500, 'slowest')
)
```

然后执行流会到达第二条 yield 语句。这个 yield 表达式会再次导致异步函数的执行被挂起，并将一个新 Promise 发给 spawn 迭代器。

```
yield new Promise(resolve => setTimeout(resolve, 200, 'slow'))
```

同样的过程会重复出现：由于还未执行到生成器函数的最后，next.done 还是 false。我们将收到的 Promise 再包装进另一个 Promise 中，以防收到的 Promise 已经被解决为 slow。接着我们再向前推进生成器函数的执行。

接着就到生成器函数的 return 语句了。此时执行流在生成器函数中被再次挂起，返回的值又传给了 spawn 迭代器。

```
return [r1, r2]
```

但此时的 next 对象如下所示。

```
{
  value: ['slowest', 'slow'],
  done: true
}
```

迭代器 spawn 检查到 next.done 确实为 true，并立即将异步函数解决为 ['slowest', 'slow']。

```
if (next.done) {
  // 成功结束，解决当前Promise
  resolve(next.value)
  return
}
```

exercise 返回的 Promise 成功兑现，最终打印出了异步函数返回的结果。

```
exercise().then(result => console.log(result))
// <- ['slowest', 'slow']
```

由此来看，异步函数不过是一个便于理解的预设罢了：在迭代生成器函数的过程中，它让值的传递尽可能畅通无阻。作为一个“语法糖”，它的背后有生成器函数、用于迭代 yield 表达式序列的 spawn 函数，以及 yield 到 await 的变换。

我们还可以从 Promise 的角度来看异步函数。思考以下示例，异步函数要等待函数调用返回的 Promise，然后再等待通过函数映射所有用户的结果。如何用 Promise 实现上述逻辑的转换呢？

```
async function getUserProfiles() {
  const users = await findAllUsers()
  const models = await Promise.all(users.map(toUserModel))
  const profiles = models.map(model => model.profile)
  return profiles
}
```

以下代码与 getUserProfiles 异步函数大致等价。注意，await 语句一般都可以改写成连缀的 Promise，而异步函数中的变量声明则可以转移到每个 Promise 的反应函数中。考虑到异步函数总是要返回 Promise，这里就原封不动地返回了。但是我们自己心里必须清楚，在将异步函数转换为 Promise 时，应该将 Promise.resolve(result) 返回给异步函数。

```
function getUserProfiles() {
  const userPromise = findAllUsers()
  const modelPromise = userPromise.then(users =>
    Promise.all(users.map(toUserModel))
    )
  const profilePromise = modelPromise.then(models =>
    models.map(model => model.profile)
  )
  return profilePromise
}
```

说到异步函数其实是包在生成器和 Promise 外的一层“语法糖”，我们一定会想到：掌握异步函数背后的这些语言构造是非常重要的。唯有如此，才能更深入地理解如何混合、搭配、联结这些异步执行机制。

4.5 异步迭代

4.2 节解释过，迭代器以 Symbol.iterator 为接口定义了应该如何迭代对象。

```
const sequence = {
  [Symbol.iterator]() {
    const items = ['i', 't', 'e', 'r', 'a', 'b', 'l', 'e']
    return {
      next: () => ({
        done: items.length === 0,
        value: items.shift()
      })
    }
  }
}
```

你应该也记得迭代 sequence 对象的方法有很多种，如扩展操作符、Array.from、for...of，等等。

```
[...sequence]
// <- ['i', 't', 'e', 'r', 'a', 'b', 'l', 'e']
Array.from(sequence)
// <- ['i', 't', 'e', 'r', 'a', 'b', 'l', 'e']

for (const item of sequence) {
  console.log(item)
  // <- 'i'
  // <- 't'
  // <- 'e'
  // <- 'r'
  // <- 'a'
  // <- 'b'
  // <- 'l'
  // <- 'e'
}
```

迭代器按照协议委托 Symbol.iterator 实例的 next 方法返回一个有 value 和 done 属性的对象。value 属性表示序列中当前的值，而 done 是一个布尔值，用于表明序列是否结束。

4.5.1 异步迭代器

异步迭代器中的协议稍有不同：next 返回一个 Promise，这个 Promise 解决后会返回包含 value 和 done 属性的对象。Promise 支持执行序列中的异步任务，即下一步会等上一步解决之后再执行。为避免重用 Symbol.iterator 可能造成的认知困惑，ES6 引入了 Symbol.asyncIterator，专门用于声明异步迭代器。

简单修改两个地方即可将 sequence 由同步可迭代对象变成异步可迭代对象：一是将 Symbol.iterator 改成 Symbol.asyncIterator，二是将 next 方法中返回的值包装在 Promise.resolve

中，这样就会返回一个 Promise 了。

```
const sequence = {
  [Symbol.asyncIterator]() {
    const items = ['i', 't', 'e', 'r', 'a', 'b', 'l', 'e']
    return {
      next: () => Promise.resolve({
        done: items.length === 0,
        value: items.shift()
      })
    }
  }
}
```

举个例子，我们可以创建一个每隔指定时间就增加一次自己值的无穷序列。以下代码中定义了一个 interval 函数来返回这个异步无穷序列，每一步都会在指定的时间 duration 过去后解决为下一个值。

```
const interval = duration => ({
  [Symbol.asyncIterator]: () => ({
    i: 0,
    next() {
      return new Promise(resolve =>
        setTimeout(() => resolve({
          value: this.i++,
          done: false
        }), duration)
      )
    }
  })
})
```

为了使用这个异步迭代器，我们可以使用随同异步迭代器一起引入的 for await..of 语法。这是将异步代码写得像同步代码的另一种方法。注意，只能在异步函数中使用 for await..of 语句。

```
async function print() {
  for await (const i of interval(1000)) {
    console.log(`${ i } seconds elapsed.`)
  }
}
print()
```

在撰写本书时，异步迭代器以及接下来要介绍的异步生成器都处于 ECMAScript 的阶段 3。

4.5.2　异步生成器

与普通迭代器一样，异步迭代器也有对应的异步生成器。异步生成器函数与生成器函数类似，区别在于前者也支持 await 和 for await..of 声明。以下示例展示了一个按固定间隔周期性获取资源的生成器 fetchInterval。

```
async function* fetchInterval(duration, ...params) {
  for await (const i of interval(duration)) {
    yield await fetch(...params)
  }
}
```

每一步执行完后，异步生成器会返回一个签名为 `{ next, return, throw }` 的对象，其方法返回的 Promise 会解决为 `{ value, done }` 形式的对象。普通生成器会直接返回 `{ value, done }` 对象。

使用异步生成器 `fetchInterval` 与使用基于对象的异步迭代器 `interval` 没什么区别。以下示例展示了如何使用 `fetchInterval` 生成器获取 HTTP 资源 `/api/status`，然后再使用返回的 JSON 响应。完成每一步后都会等 1 秒再重复相同过程。

```
async function process() {
  for await (const response of fetchInterval(
  1000,
  '/api/status'
)) {
    const data = await response.json()
    // 使用更新后的数据
  }
}
process()
```

正如 4.2.2 节提到的，为避免无穷循环，根据某些条件中断这些序列是非常重要的。

第 5 章

巧妙使用ES中的集合

JavaScript 数据结构很灵活，任何对象都可以转换为散列映射（hash-map），其中键由字符串构成，可以映射到任意值。 例如，我们可以用一个对象将 npm 包名映射到它们的元数据，如下所示。

```
const registry = {}
function set(name, meta) {
  registry[name] = meta
}
function get(name) {
  return registry[name]
}
set('contra', { description: 'Asynchronous flow control' })
set('dragula', { description: 'Drag and drop' })
set('woofmark', { description: 'Markdown and WYSIWYG editor' })
```

这种方法有以下几个问题。

- 用户使用 __proto__、toString 或 Object.prototype 中的任意属性作为键值会产生安全问题，并会对使用这个对象造成麻烦。
- 使用 for..in 进行迭代需要依赖 Object#hasOwnProperty 来确保属性不是由继承而来。
- 使用 Object.keys(registry).forEach 对列表进行迭代过于烦琐。
- 键仅限于字符串，而不能是 DOM 元素或其他非字符串。

第一个问题可以通过使用前缀来解决，但要注意，无论取值还是赋值，都要通过函数添加前缀才能避免出错。

```
const registry = {}
function set(name, meta) {
  registry['pkg:' + name] = meta
}
function get(name) {
  return registry['pkg:' + name]
}
```

另外，我们还可以使用 `Object.create(null)` 而非空对象字面量。 在这种情况下，创建的对象不会继承 `Object.prototype`，这意味着它不会受到 `__proto__` 等内置属性的影响。

```
const registry = Object.create(null)
function set(name, meta) {
  registry[name] = meta
}
function get(name) {
  return registry[name]
}
```

对于迭代，我们可以创建一个返回键 / 值对元组的 `list` 函数。

```
const registry = Object.create(null)
function list() {
  return Object.keys(registry).map(key => [key, registry[key]])
}
```

我们也可以在散列映射上实现迭代器协议。为了方便使用，这里采用了一种复杂的实现方式：以下示例中使用迭代器的代码比前面使用了 `Object.keys` 和 `Array#map` 方法的 `list` 函数更难于理解。然而，遵循迭代器协议意味着不需要自定义 `list` 函数，从而能够更简便地访问列表。

```
const registry = Object.create(null)
registry[Symbol.iterator] = () => {
  const keys = Object.keys(registry)
  return {
    next() {
      const done = keys.length === 0
      const key = keys.shift()
      const value = [key, registry[key]]
      return { done, value }
    }
  }
}
console.log([...registry])
```

在 ES5 代码中无法使用非字符串键。好在 ES6 集合为我们提供了更好的解决方案。ES6 集合没有键命名问题，并且改善了集合行为，比如，上面示例中自定义散列映射的迭代器已经成为了 ES6 中的内置方法。与此同时，ES6 集合允许使用任意值作为键，不像常规 JavaScript 对象那样仅限于字符串键。

我们来探讨一下它们的实际使用和内部机制。

5.1 使用ES6 map

ES6 引入了 Map 等内置集合，减少了手动构建散列映射的工作量。Map 是 ES6 中的一个键 / 值数据结构，它使得在 JavaScript 中创建映射变得更加自然且高效，而不再需要对象字面量。

5.1.1 初识ES6 map

接下来我们用 Map 重写前面的代码。如你所见，代码少了很多，因为用 ES5 构建自定义散列映射的实现细节已经内置到 Map 中了。

```
const map = new Map()
map.set('contra', { description: 'Asynchronous flow control' })
map.set('dragula', { description: 'Drag and drop' })
map.set('woofmark', {
  description: 'Markdown and WYSIWYG editor'
})
console.log([...map])
```

创建 map 后，就可以通过给 map.has 方法传入一个 key 来查询它是否包含某个成员。

```
map.has('contra')
// <- true
map.has('jquery')
// <- false
```

传统的对象会对键进行类型转换，但 map 不会。显然，这是一个优点，但需要注意的是，在查询 map 时，map 对键的处理方式与传统对象也不同。以下示例在创建 Map 实例时传入了一个可迭代的键 / 值对数组，可以看出键值并没有转换为字符串。

```
const map = new Map([[1, 'the number one']])
map.has(1)
// <- true
map.has('1')
// <- false
```

map.get 方法接收一个参数，如果存在该键，则返回对应的值。

```
map.get('contra')
// <- { description: 'Asynchronous flow control' }
```

传入想要删除的键，就可以通过 map.delete 方法从 map 中删除对应的值。

```
map.delete('contra')
map.get('contra')
// <- undefined
```

你可以通过 map.clear 清空整个 Map，但不会丢失对 Map 本身的引用。这很适合用来重置对象的状态。

```
const map = new Map([[1, 2], [3, 4], [5, 6]])
map.has(1)
// <- true
map.clear()
map.has(1)
// <- false
[...map]
// <- []
```

map 有一个只读的 `.size` 属性，该属性与 `Array#length` 类似，用于表示当前 map 中的成员总数。

```
const map = new Map([[1, 2], [3, 4], [5, 6]])
map.size
// <- 3
map.delete(3)
map.size
// <- 2
map.clear()
map.size
// <- 0
```

任意对象都可以作为 map 的键：我们不仅可以使用符号、数值或字符串这样的基本类型，还可以使用函数、对象、日期，甚至 DOM 元素。这些键不会像在普通的 JavaScript 对象中那样被转换为字符串，而是会保留它们的引用。

```
const map = new Map()
map.set(new Date(), function today() {})
map.set(() => 'key', { key: 'door' })
map.set(Symbol('items'), [1, 2])
```

如果选择使用符号作为 map 的键，那么必须使用对同一符号的引用来获取该成员，如下所示。

```
const map = new Map()
const key = Symbol('items')
map.set(key, [1, 2])
map.get(Symbol('items')) // 与变量key的引用不同
// <- undefined
map.get(key)
// <- [1, 2]
```

假设有一个由键 / 值对构成的二维数组 `items`，我们可以使用 `for..of` 来迭代 `items`，并用 `map.set` 将每个键 / 值对添加到 map 中，如下所示。注意，我们在 `for..of` 循环中使用了解构来轻松地取出 `key` 和 `value`。

```
const items = [
  [new Date(), function today() {}],
  [() => 'key', { key: 'door' }],
  [Symbol('items'), [1, 2]]
```

```
]
const map = new Map()
for (const [key, value] of items) {
  map.set(key, value)
}
```

map 也是可迭代对象，因为它实现了 Symbol.iterator 方法。因此，就像前面迭代数组创建 map 那样，我们也可以使用 for..of 循环通过迭代 map 来创建副本。

```
const copy = new Map()
for (const [key, value] of map) {
  copy.set(key, value)
}
```

为简单起见，你可以使用任何遵循可迭代协议的对象直接初始化 map，并生成 [key, value] 的集合。以下代码段用数组创建了一个新的 Map。在这种情况下，所有的迭代行为都发生在 Map 的构造函数中。

```
const items = [
  [new Date(), function today() {}],
  [() => 'key', { key: 'door' }],
  [Symbol('items'), [1, 2]]
]
const map = new Map(items)
```

创建 map 的副本甚至更容易：将要复制的 map 作为参数传递给新 map 的构造函数就可以获取副本。这里并没有重载 new Map(map)。我们利用 map 本身既可以实现可迭代协议，又可以将可迭代对象作为参数的特性来创建新的 map。以下代码展示了这一过程是多么简单。

```
const copy = new Map(map)
```

因为 map 是可迭代对象，所以我们可以很容易地将它传入其他 map 中，也可以很轻松地使用它们。以下代码展示了如何使用扩展运算符来访问 map。

```
const map = new Map()
map.set(1, 'one')
map.set(2, 'two')
map.set(3, 'three')
console.log([...map])
// <- [[1, 'one'], [2, 'two'], [3, 'three']]
```

以下代码结合了 ES6 中的 Map、for..of 循环、let 变量和模板字符串这些新特性。

```
const map = new Map()
map.set(1, 'one')
map.set(2, 'two')
map.set(3, 'three')
for (const [key, value] of map) {
```

```
  console.log(`${ key }: ${ value }`)
  // <- '1: one'
  // <- '2: two'
  // <- '3: three'
}
```

虽然需要通过 API 才能访问 map 成员，但与散列映射相同，它们的键是唯一的。不断给同一个键赋值只会覆写其对应的值。以下代码段表明，即使反复为 'a' 赋值，map 中也只会包含一个成员。

```
const map = new Map()
map.set('a', 1)
map.set('a', 2)
map.set('a', 3)
console.log([...map])
// <- [['a', 3]]
```

ES6 map 使用 SameValueZero 算法对键进行比较，其中 NaN 等于 NaN，-0 等于 +0。以下代码表明，即便 NaN 通常不等于自身，但作为 Map 的一个键，NaN 始终代表一个常量。

```
console.log(NaN === NaN)
// <- false
console.log(-0 === +0)
// <- true
const map = new Map()
map.set(NaN, 'one')
map.set(NaN, 'two')
map.set(-0, 'three')
map.set(+0, 'four')
console.log([...map])
// <- [[NaN, 'two'], [0, 'four']]
```

迭代 Map 实际上是在遍历它的 .entries()。因为 map[Symbol.iterator] 指向 map.entries，所以我们不必显式地遍历 .entries()。.entries() 方法返回 map 中键 / 值对的迭代器。

```
console.log(map[Symbol.iterator] === map.entries)
// <- true
```

我们还可以使用另外两个 Map 迭代器：.keys() 和 .values()。前者枚举了 map 中所有的键，后者枚举了所有的值，而 .entries() 则枚举了所有的键 / 值对。以下代码展示了这三种方法之间的区别。

```
const map = new Map([[1, 2], [3, 4], [5, 6]])
console.log([...map.keys()])
// <- [1, 3, 5]
console.log([...map.values()])
// <- [2, 4, 6]
console.log([...map.entries()])
// <- [[1, 2], [3, 4], [5, 6]]
```

对 map 中成员的迭代遵循插入时的顺序，这与 `Object.keys` 不同，后者遵循任意顺序。但在实践中，插入顺序往往由 JavaScript 引擎保存，与规范无关。

map 的 `.forEach` 方法与 ES5 中 `Array` 对象的该方法用法相同，签名为 `(value, key, map)`，其中 `value` 和 `key` 表示迭代中的当前元素的值和键，`map` 表示被迭代的 map。再次强调，`Map` 中的键是不会转换为字符串的，如下所示。

```
const map = new Map([
  [NaN, 1],
  [Symbol(), 2],
  ['key', 'value'],
  [{ name: 'Kent' }, 'is a person']
])
map.forEach((value, key) => console.log(key, value))
// <- NaN 1
// <- Symbol() 2
// <- 'key' 'value'
// <- { name: 'Kent' } 'is a person'
```

至此，我们了解了 `Map` 的键可以是任何对象的引用。接下来我们看看这个 API 的具体用法。

5.1.2 散列映射和DOM元素

在 ES5 中，如果想要将 DOM 元素与（连接该元素与某个库的）API 对象相关联，就不得不用一种冗余又低效的方式来实现，如下所示。以下代码将给定 DOM 元素与一个含有若干方法的 API 对象相关联，并将它们存储在 map 中，之后我们就可以找出 DOM 元素对应的 API 对象了。

```
const map = []
function customThing(el) {
  const mapped = findByElement(el)
  if (mapped) {
    return mapped
  }
  const api = {
    // 自定义API方法
  }
  const entry = storeInMap(el, api)
  api.destroy = destroy.bind(null, entry)
  return api
}
function storeInMap(el, api) {
  const entry = { el, api }
  map.push(entry)
  return entry
}
function findByElement(query) {
  for (const { el, api } of map) {
    if (el === query) {
```

```
      return api
    }
  }
}
function destroy(entry) {
  const index = map.indexOf(entry)
  map.splice(index, 1)
}
```

Map 最有价值的特性之一是可以通过任何对象进行索引，如 DOM 元素。结合 Map 处理集合的能力，很多事情都可以简化，这在 jQuery 这样重 DOM 操作的代码库中很重要，DOM 元素经常需要映射到这些库的内部状态。

以下示例展示了 Map 是如何减轻维护压力的。

```
const map = new Map()
function customThing(el) {
  const mapped = findByElement(el)
  if (mapped) {
    return mapped
  }
  const api = {
    //自定义API方法
    destroy: destroy.bind(null, el)
  }
  storeInMap(el, api)
  return api
}
function storeInMap(el, api) {
  map.set(el, api)
}
function findByElement(el) {
  return map.get(el)
}
function destroy(el) {
  map.delete(el)
}
```

使用原生 Map 方法仅需一行代码即可实现映射操作，这意味着我们可以内联这些函数，不必担心可读性问题。以下代码是前面那段 ES5 代码的简化版本。这里不再关心实现细节，而是真正回归从 DOM 到 API 映射的本质需求。

```
const map = new Map()
function customThing(el) {
  const mapped = map.get(el)
  if (mapped) {
    return mapped
  }
  const api = {
    // 自定义API方法
    destroy: () => map.delete(el)
  }
```

```
  map.set(el, api)
  return api
}
```

然而，`Map` 并不是 ES6 中唯一的内置集合，还有 `WeakMap`、`Set` 和 `WeakSet`。接下来继续讨论 `WeakMap`。

5.2 理解和使用WeakMap

大部分情况下，你可以视 `WeakMap` 为 `Map` 的子集。相比于 `Map`，`WeakMap` 有着更为精简的 API。因为 `WeakMap` 中没有迭代器协议，所以使用 `WeakMap` 创建的集合不像 `Map` 那样可迭代，这意味着没有 `WeakMap#entries`、`WeakMap#keys`、`WeakMap#values`、`WeakMap#forEach` 及 `WeakMap#clear` 等方法。

另一个区别是，`Map` 中的 `key` 可以是对象引用或其他，但 `WeakMap` 中的每个 `key` 都必须是一个对象。记住，`Symbol` 是一种值类型，因此也不能作为 `WeakMap` 的 `key`。

```
const map = new WeakMap()
map.set(Date.now, 'now')
map.set(1, 1)
// <- TypeError
map.set(Symbol(), 2)
// <- TypeError
```

为了成为一个特征有限的集合，`WeakMap` 中键的引用是弱保持的，也就是说，如果作为 `WeakMap` 键的对象除了弱引用外没有其他的引用，则该对象将被垃圾回收清除。举个例子，当一个 `person` 对象存有一些元数据时，如果你希望这个 `person` 对象在对它的唯一引用是这些关联的元数据时被垃圾回收清除，那么 `WeakMap` 的这一特性就有了用武之地。现在你可以使用 `person` 作为键将这些元数据保存在 `WeakMap` 中。

从这个意义上讲，当维护 `WeakMap` 的组件不拥有映射对象，但希望将它自己的信息分配给它们而无须修改原始对象或它们的生命周期时，`WeakMap` 最有用。例如，当从文档中删除 DOM 节点时，回收内存。

通过向构造函数传入一个可迭代对象，可以初始化一个 `WeakMap`。与创建一个 `Map` 相同，这个可迭代对象应该是一个键 / 值对组成的列表。

```
const map = new WeakMap([
  [new Date(), 'foo'],
  [() => 'bar', 'baz']
])
```

为了高效地实现弱引用，`WeakMap` 的 API 更少，但它仍然保有与 `Map` 相同的 `.has`、`.get` 和 `.delete` 方法。以下代码展示了这些方法的用法。

```
const date = new Date()
const map = new WeakMap([[date, 'foo'], [() => 'bar', 'baz']])
map.has(date)
// <- true
map.get(date)
// <- 'foo'
map.delete(date)
map.has(date)
// <- false
```

WeakMap是一个糟糕的Map吗

令人产生误解的根源在于它的名字，`WeakMap` 中的“weak”表示它对其键是弱引用的，如果除了用作 `WeakMap` 键之外，作为键的对象没有其他引用，那么它们就会被垃圾回收清除。这与 `Map` 形成了鲜明的对比，`Map` 中的键对象是强引用的，可以阻止 `Map` 键和值被垃圾回收清除。

因此，我们可以用 `WeakMap` 来指定元数据或扩展对象，同时在没有其他引用的情况下回收该对象。Node.js 中 `process.on('unhandledRejection')` 的底层实现就是一个完美的示例，它使用 `WeakMap` 来跟踪尚未处理的被拒绝的 Promise。`WeakMap` 不会紧紧抓住与这些 Promise 相关的状态，因此可以用 `WeakMap` 来阻止内存泄漏。这样一个简单的 map 既可以较弱地保持状态，又可以灵活地在 Promise 不再被引用时将其从 map 中删除。

如果想要保存 DOM 元素的数据并在不需要时从内存中释放，我们也可以使用 `WeakMap`。就这一点而言，在实现与 DOM 相关的 API 缓存解决方案时，使用 `WeakMap` 比我们早先使用 `Map` 更好。

总而言之，`WeakMap` 当然不比 `Map` 差，只是尺有所短、寸有所长罢了。

5.3 ES6中的Set

`Set` 是 ES6 中内置的新集合类型，用于表示一组值。从某些方面来说，`Set` 与 `Map` 类似。

- `Set` 也是可迭代的。
- `Set` 构造函数也接受一个可迭代对象。
- `Set` 也有一个 `.size` 属性。
- 与 `Map` 中的键一样，`Set` 中的值可以是任意值或对象引用。
- 与 `Map` 中的键一样，`Set` 中的值必须是唯一的。
- `NaN` 在 `Set` 中也等于 `NaN`。
- `Set` 同样拥有 `.keys`、`.values`、`.entries`、`.forEach`、`.has`、`.delete` 和 `.clear` 方法。

然而，`Set` 与 `Map` 在几个重要的方面有所不同。`Set` 没有键 / 值对，它只有一个维度。你可以将 `Set` 视为元素彼此不同的数组。

Set 中没有 .get 方法。在仅有的一个维度中无法获取除了值以外的其他东西，因此 set.get(value) 方法是多余的。如果想要检查一个 value 是否在 Set 中，可以使用 set.has(value)。

同样，set.set(value) 方法也没有意义，因为我们不能为 value 设置 key，而是仅向该 Set 添加值。因此，将值添加到 Set 的方法是 set.add，如下所示。

```
const set = new Set()
set.add({ an: 'example' })
```

Set 是可迭代的，但与 Map 不同，Set 只能迭代值，而不能迭代键 / 值对。以下示例展示了如何使用扩展运算符通过数组创建 Set，并将 Set 展开为一个一维列表。

```
const set = new Set(['a', 'b', 'c'])
console.log([...set])
// <- ['a', 'b', 'c']
```

在下面的示例中，你会注意到 Set 不包含重复成员，每个元素必须是唯一的。

```
const set = new Set(['a', 'b', 'b', 'c', 'c'])
console.log([...set])
// <- ['a', 'b', 'c']
```

以下代码首先创建了一个包含页面上所有 <div> 元素的 Set，并打印元素个数。然后查询 DOM 并调用 set.add 将这些 DOM 元素再次加入 set 中。鉴于这些 DOM 元素已经在 set 中，.size 属性不会改变，这意味着 set 也没有改变。

```
function divs() {
  return document.querySelectorAll('div')
}
const set = new Set(divs())
console.log(set.size)
// <- 56
divs().forEach(div => set.add(div))
console.log(set.size)
// <- 56
```

因为 Set 没有键，所以 Set#entries 方法为集合中的每个元素返回 [value，value] 的迭代器。

```
const set = new Set(['a', 'b', 'c'])
console.log([...set.entries()])
// <- [['a', 'a'], ['b', 'b'], ['c', 'c']]
```

Map#entries 返回 [key， value] 的迭代器，Set#entries 方法也很类似。使用 Set#entries 作为 Set 集合的默认迭代器并不是很有价值，因为通常用法是在 for..of 或 Array.from 中展开 set。在这些情况下，你可能想迭代的是集合中的一系列 value，而不是一系列 [value，value]。

因此，Set 默认的迭代器是 Set#values，而不是像 Map 那样默认的迭代器为 Map#entries，如下所示。

```
const map = new Map()
console.log(map[Symbol.iterator] === map.entries)
// <- true
const set = new Set()
console.log(set[Symbol.iterator] === set.entries)
// <- false
console.log(set[Symbol.iterator] === set.values)
// <- true
```

为保持一致性，`Set#keys` 方法也返回 `value` 的迭代器，实际上它是对 `Set#values` 迭代器的引用。

```
const set = new Set()
console.log(set.keys === set.values)
// <- true
```

5.4 ES6 WeakSet

就像 `Map` 和 `WeakMap` 的关系一样，`WeakSet` 是 `Set` 的弱版本，它无法迭代。`WeakSet` 中的值必须是唯一的对象引用。如果 `WeakSet` 中的值没有其他引用，那么它将被垃圾回收。

`WeakSet` 只有 `.add`、`.delete` 以及检查其中是否有给定值的 `.has` 方法。与 `Set` 一样，`WeakSet` 也没有 `.get` 方法，因为它是一维的。

与 `WeakMap` 一样，我们也不能向 `WeakSet` 添加字符串或符号这样的基本值。

```
const set = new WeakSet()
set.add('a')
// <- TypeError
set.add(Symbol())
// <- TypeError
```

虽然 `WeakSet` 本身不可迭代，但可以将迭代器传递给构造函数。在构造集合时，可以对可迭代对象进行迭代，并将可迭代序列中的每个成员添加到集合中，如下所示。

```
const set = new WeakSet([
  new Date(),
  {},
  () => {},
  [1]
])
```

通过使用 `WeakSet`，我们可以确保 `Car` 类中的方法是 `Car` 类的实例对象调用的，如下所示。

```
const cars = new WeakSet()
class Car {
  constructor() {
    cars.add(this)
  }
```

```
  fuelUp() {
    if (!cars.has(this)) {
      throw new TypeError('Car#fuelUp called on a non-Car!')
    }
  }
}
```

举个更有用的例子，考虑以下的 `istOwnProperties` 接口，该接口实现的是打印传入对象的包括子属性在内的所有属性。这个接口应该知道如何处理循环引用，而不是陷入无限循环中。你会如何实现这样一个 API 呢？

```
const circle = { cx: 20, cy: 5, r: 15 }
circle.self = circle
listOwnProperties({
  circle,
  numbers: [1, 5, 7],
  sum: (a, b) => a + b
})
// <- circle.cx: 20
// <- circle.cy: 5
// <- circle.r: 15
// <- circle.self: [circular]
// <- numbers.0: 1
// <- numbers.1: 5
// <- numbers.2: 7
// <- sum: (a, b) => a + b
```

其中一种实现方案是在 `WeakSet` 中保存一个可见引用列表，以避免非线性查找。这里使用 `WeakSet` 而不是 `Set`，因为我们不需要 `Set` 中存在的其他功能。

```
function listOwnProperties(input) {
  recurse(input)

  function recurse(source, lastPrefix, seen = new WeakSet()) {
    Object.keys(source).forEach(printOrRecurse)

    function printOrRecurse(key) {
      const value = source[key]
      const prefix = lastPrefix
        ? `${ lastPrefix }.${ key }`
          : key
      const shouldRecur = (
        isObject(value) ||
        Array.isArray(value)
      )
      if (shouldRecur) {
        if (!seen.has(value)) {
          seen.add(value)
          recurse(value, prefix, seen)
        } else {
          console.log(`${ prefix }: [circular]`)
        }
```

```
        } else {
          console.log(`${ prefix }: ${ value }`)
        }
      }
    }
  }
  function isObject(value) {
    return Object.prototype.toString.call(value) ===
      '[object Object]'
  }
```

更常见的用法是保存 DOM 元素。在一个 DOM 库中，通常我们需要在第一次与一个 DOM 元素互动时操作它，但又不想留下痕迹，此时就可以使用 `WeakSet`。例如，库想要为 `target` 元素添加一些子节点，但又无法确保这些子节点是否被添加过，同时也不想弄乱这个 `target`；或者它只想在第一次调用时执行某种操作。

```
const elements = new WeakSet()
function dommy(target) {
  if (elements.has(target)) {
    return
  }
  elements.add(target)
  // 执行工作……
})
```

无论何种原因，如果希望在不显式地改变 DOM 元素的情况下，用标记与 DOM 元素保持关联，`WeakSet` 是一条可行的路。如果不想使用简单的标记来关联任意数据，或许你可以试试 `WeakMap`。在决定是否使用 `Map`、`WeakMap`、`Set` 或 `WeakSet` 之前，你应该问自己一系列的问题。例如，如果需要保留与对象相关的数据，那么可以考虑使用弱集合。如果只关心某个元素是否存在，那么可能需要使用 `Set`。如果想要创建一个缓存，建议使用 `Map`。

针对此前实现起来很烦琐的用例（如前面 `Map` 的例子），或者难以正确执行的用例（如 `WeakMap` 的例子，其中当不再需要引用时将其清除，以避免内存泄漏），ES6 的集合都提供了内置的解决方案。

第 6 章

使用代理控制属性访问

代理（proxy）是 ES6 中受人瞩目又强大的一个属性，它在 API 使用者和对象之间扮演了一个中间人的角色。简而言之，我们可以使用 `Proxy` 来控制被访问的底层 `target` 对象的属性的行为。`handler` 对象可以用于配置 `Proxy` 的捕获器（trap），后者用于定义和限制访问底层对象的方式。

6.1　了解代理

默认情况下，代理不会做什么，事实上它什么也不做。如果不提供任何配置，那么代理就像是通向 `target` 对象的一个通道，也就是所谓的“无操作转发代理”。换句话说，对代理对象进行的所有操作都会传递到底层对象。

我们在以下代码中创建了一个无操作转发 `Proxy`。可以看到，通过给 `proxy.exposed` 赋值，我们可以将该值传递到 `target.exposed`。我们可以将代理想象为其底层对象的门卫：它们可以允许某些操作通过，也可以拒绝另一些操作通过，但无论允许还是拒绝，都是有充分理由的。

```
const target = {}
const handler = {}
const proxy = new Proxy(target, handler)
proxy.exposed = true
console.log(target.exposed)
// <- true
console.log(proxy.somethingElse)
// <- undefined
```

通过添加捕获器，我们可以使代理对象发挥更大的作用。捕获器可以从不同角度拦截对 target 对象的操作，前提是这些操作必须通过 proxy 进行。比如，我们可以通过 get 捕获器来打印每次从 target 对象取得的值，或者通过 set 捕获器阻止对某些属性的写操作。我们先从 get 捕获器开始。

6.1.1 捕获get访问

以下代码中的 proxy 对象可以捕获对 target 对象的每一次访问，因为我们为它定义了一个 handler.get 捕获器。我们可以通过这个捕获器转换任意属性的值，然后再将转换结果返回给访问者。

```
const handler = {
  get(target, key) {
    console.log(`Get on property "${ key }"`)
    return target[key]
  }
}
const target = {}
const proxy = new Proxy(target, handler)
proxy.numbers = [1, 1, 2, 3, 5, 8, 13]
proxy.numbers
// 'Get on property "numbers"'
// <- [1, 1, 2, 3, 5, 8, 13]
proxy['something-else']
// 'Get on property "something-else"'
// <- undefined
```

作为代理的补充，ES6 引入了一个内置的 Reflect 对象。ES6 代理的捕获器会一对一地映射到 Reflect 对象的 API。具体来说，每个捕获器在 Reflect 对象上都有一个对应的反射方法。正是因为有了这些方法，我们才可以使用代理捕获器的默认行为，而无须自己来实现。

在以下示例中，我们使用 Reflect.get 实现了 get 操作的默认行为，而无须自己再编写访问 target 对象中的 key 属性的代码。虽然眼前这个示例没什么值得大惊小怪的，但如果真要实现其他捕获器的默认行为，可能就没那么简单，也没那么容易写正确了。我们可以将捕获器的所有参数都传递给相应的反射 API，然后返回结果。

```
const handler = {
  get(target, key) {
    console.log(`Get on property "${ key }"`)
    return Reflect.get(target, key)
  }
}
const target = {}
const proxy = new Proxy(target, handler)
```

get 捕获器不一定总是返回 target[key] 的原始值。比如，要想确保所有加了前缀“_”的属性都不能访问，此时可以抛出错误，以便使用者知道这些属性不能通过代理访问。

```
const handler = {
  get(target, key) {
    if (key.startsWith('_')) {
      throw new Error(`Property "${ key }" is inaccessible.`)
    }
    return Reflect.get(target, key)
  }
}
const target = {}
const proxy = new Proxy(target, handler)
proxy._secret
// <- Uncaught Error: Property "_secret" is inaccessible.
```

你可能已经意识到了，借助代理并定义明确的规则来禁止访问 target 对象的某些属性，对外只暴露代理而不暴露 target 对象，应该是代理最主要的用途。这样一来，我们仍然可以自由地访问底层对象，但使用者必须通过代理并遵守访问规则。也就是说，如何访问对象由我们说了算。这在 ES6 引入代理前是不可能做到的。

6.1.2 捕获set访问

与 get 捕获器相对，set 捕获器可以拦截属性赋值。假设想要阻止对下划线开头的属性的赋值，我们可以像实现前面的 get 捕获器那样通过 set 捕获来达到目的。

通过 proxy 访问 target 时，以下示例中的 Proxy 会同时阻止以下划线开头的属性的 get 和 set 操作。注意其中的 set 捕获器应该在什么情况下返回 true。返回 true 意味着给属性 key 设置 value 应该成功了。如果 set 捕获器的返回值是 false，则设置相应属性的值在严格模式下会导致抛出 TypeError，否则会静默失败。如果这里使用 Reflect.set（像前面介绍的那样），那么我们就不需要自己考虑这些实现细节了。只要返回 Reflect.set(target, key, value) 即可。这样一来，如果以后有人阅读我们的代码，就会明白我们在这里使用了 Reflect.set，这等价于默认操作，即等价于不使用 Proxy 的情形。

```
const handler = {
  get(target, key) {
    invariant(key, 'get')
    return Reflect.get(target, key)
  },
  set(target, key, value) {
    invariant(key, 'set')
    return Reflect.set(target, key, value)
  }
}
function invariant(key, action) {
  if (key.startsWith('_')) {
    throw new Error(`Can't ${ action } private "${ key }"
    property`)
  }
}
const target = {}
const proxy = new Proxy(target, handler)
```

以下代码演示了使用者通过 proxy 对象操作 target 的情形。

```
proxy.text = 'the great black pony ate your lunch'
console.log(target.text)
// <- 'the great black pony ate your lunch'
proxy._secret
// <- Error: Can't get private "_secret" property
proxy._secret = 'invalidate'
// <- Error: Can't set private "_secret" property
```

对使用者而言，前面示例中被代理的 target 对象应该是完全隐藏的，这样才能强迫他们必须通过 proxy 来实现操作。阻止对 target 的直接访问意味着使用者必须遵守 proxy 对象定义的访问规则，如“**禁止访问以下划线开头的属性**”。

最终，我们可以将被代理的对象包装在一个函数中，然后返回 proxy。

```
function proxied() {
  const target = {}
  const handler = {
    get(target, key) {
      invariant(key, 'get')
      return Reflect.get(target, key)
    },
    set(target, key, value) {
      invariant(key, 'set')
      return Reflect.set(target, key, value)
    }
  }
  return new Proxy(target, handler)
}
function invariant(key, action) {
  if (key.startsWith('_')) {
    throw new Error(`Can't ${ action } private "${ key }"
    property`)
  }
}
```

用法相同，只不过现在对 target 的访问完全由 proxy 及其捕获器接管。此时，通过 proxy 访问 target 中类似 _secret 的属性是完全不可能的，因为在 proxied 函数外部根本访问不到 target。对使用者而言，target 被永远封存了。

为此，我们可以编写一个通用的代理函数来接收一个 orginal 对象，并返回一个 proxy 对象。然后我们就可以在需要暴露公共 API 时调用它，如下所示。这里的 concealWithPrefix 函数将 orginal 对象包装在一个 Proxy 中，而且以 prefix 值（未提供则为 _）为前缀的属性都不能访问。

```
function concealWithPrefix(original, prefix='_') {
  const handler = {
    get(original, key) {
```

```
      invariant(key, 'get')
      return Reflect.get(original, key)
    },
    set(original, key, value) {
      invariant(key, 'set')
      return Reflect.set(original, key, value)
    }
  }
  return new Proxy(original, handler)
  function invariant(key, action) {
    if (key.startsWith(prefix)) {
      throw new Error(`Can't ${ action } private "${ key }"
      property`)
    }
  }
}
const target = {
  _secret: 'secret',
  text: 'everyone-can-read-this'
}
const proxy = concealWithPrefix(target)
// 将代理暴露给使用者
```

你可能会说，在 ES5 中使用对 concealWithPrefix 函数私有的变量也可以达到相同目的，根本不必使用 Proxy。二者的区别是，使用 Proxy 可以实现动态的属性私有化。如果不使用 Proxy，就不能将所有以下划线开头的属性标记为私有。虽然可以在对象上使用 Object.freeze[1]，但之后不光是使用者，就连你自己也不能修改属性了。或者你可以给每个属性都定义 getter 和 setter 访问器，然而这样不能阻止对所有属性的访问，只能阻止那些明确配置了 getter 和 setter 的属性。

6.1.3 通过代理实现模式验证

有时我们需要验证基于用户输入生成的对象。作为模型，这个对象遵循某种模式（schema）：结构是什么样的，应该有哪些属性，属性应该是什么类型，以及应该如何填充这些属性，等等。比如，我们可以验证一个电子邮件字段 customer 包含邮件地址，一个数值类型的 cost 字段包含数值，一个必填的 name 字段没有缺失。

实现模式验证的方法有很多。我们可以将模型传给一个验证函数，如果在对象上发现无效值，则抛出错误。但在确认它有效后，我们必须确保对象不会再被修改。我们可以逐个验证每个属性，但必须记得在它们改变时再次验证。我们也可以使用 Proxy。通过给使用者提供实际模型对象的代理，我们可以确保对象永远不会处于无效状态，因为无效就会导致抛出异常。

注 1：Object.freeze 方法会阻止添加、删除属性以及修改属性值的引用。但它不会让属性值本身变得不能修改。这些值的属性仍然可以改变，前提是没有对这些对象调用 Object.freeze。

通过 Proxy 实现验证的另一个好处是，它可以帮助我们将验证从 target 对象分离出来，而验证往往发生在比较“恶劣”的环境下。target 对象会保持纯粹 JavaScript 对象的状态，因为你提供给使用者的只是验证代理，而代理可以确保底层对象始终有效，不被污染。

与验证函数类似，代理的处理逻辑也可以在不同的 Proxy 间重复使用，而无须依赖原型继承或 ES6 的类。

以下示例定义了一个简单的 validator 对象，它包含一个 set 捕获器，用于查询映射中的键。通过代理设置属性时，就会在映射中查询相应的键是否存在。如果映射给该属性设定了规则，那么它就会运行相应的函数来判定相应的赋值是否有效。只要用 validator 的代理设置 person 的属性，就可以按照预定义的验证规则确保模型符合要求。

```
const validations = new Map()
const validator = {
  set(target, key, value) {
    if (validations.has(key)) {
      return validations[key](value)
    }
    return Reflect.set(target, key, value)
  }
}
validations.set('age', validateAge)

function validateAge(value) {
  if (typeof value !== 'number' || Number.isNaN(value)) {
    throw new TypeError('Age must be a number')
  }
  if (value <= 0) {
    throw new TypeError('Age must be a positive number')
  }
  return true
}
```

以下代码展示了如何将 validator 作为处理器使用。这是一个通用的代理处理器，我们将它传给 Proxy 以验证 person 对象。然后处理器会对任何通过代理设置的属性按照验证规则进行验证，确保设置的值满足规则要求。这里只添加了一条规则，即 age 必须是一个大于零的正数。

```
const person = {}
const proxy = new Proxy(person, validator)
proxy.age = 'twenty three'
// <- TypeError: Age must be a number
proxy.age = NaN
// <- TypeError: Age must be a number
proxy.age = 0
// <- TypeError: Age must be a positive number
proxy.age = 28
console.log(person.age)
// <- 28
```

代理提供了前所未有的细粒度控制，通过预定义规则决定了使用者能做什么，不能做什么。实际上，代理还有一个更加严苛的变体以允许我们完全切断对 `target` 的访问：可撤销代理。

6.2 可撤销代理

相比于 `Proxy` 对象，可撤销代理提供了更加细粒度的控制。但 API 稍有不同，一是不再使用 `new` 关键字（相比于 `new Proxy(target, handler)`），二是返回的是一个 `{ proxy, revoke }` 对象（而不仅仅是一个 `proxy` 对象）。调用 `revoke()` 后，任何操作都会导致 `proxy` 抛出错误。

接下来仍然以“仅转发的” `Proxy` 为例，我们将它改造成一个可撤销代理。注意，创建代理时没有使用 `new`，而且反复调用 `revoke()` 没有任何反应，但之后再想访问底层对象会导致错误。

```
const target = {}
const handler = {}
const { proxy, revoke } = Proxy.revocable(target, handler)
proxy.isUsable = true
console.log(proxy.isUsable)
// <- true
revoke()
revoke()
revoke()
console.log(proxy.isUsable)
// <- TypeError: illegal operation attempted on a revoked proxy
```

有时这种代理非常有用，因为你可以完全收回使用者访问代理的权限。你可以对外暴露一个可撤销代理，而将其 `revoke` 方法保存在一个 `WeakMap` 集合中。等到明显不该让使用者再访问底层对象（甚至代理）时，就可以调用 `.revoke()` 来收回它们的访问权了。

以下示例展示了两个函数，其中 `getStorage` 返回用于访问 `storage` 的代理对象，同时用 `WeakMap` 集合保存了代理及其 `revoke` 函数。当我们想要切断对 `storage` 的访问时，`revokeSorage` 将调用对应的 `revoke` 函数并从 `WeakMap` 移除该项。注意，这里允许使用者同时访问两个函数不会造成安全问题：只要通过代理访问 `storage` 的权限被收回，就无法再恢复。

```
const proxies = new WeakMap()
const storage = {}

function getStorage() {
  const handler = {}
  const { proxy, revoke } = Proxy.revocable(storage, handler)
  proxies.set(proxy, { revoke })
  return proxy
}
```

```
function revokeStorage(proxy) {
  proxies.get(proxy).revoke()
  proxies.delete(proxy)
}
```

考虑到 revoke 与 handler 捕获器同在一个作用域中，你可以制定严厉的访问规则。比如，要是使用者企图多次访问某个私有属性，则立即撤销其对 proxy 的访问权。

6.3 代理捕获器

代理最强大的地方主要体现在其各种捕获器上。除了 get 和 set，我们还可以通过代理拦截很多访问对象的操作。

前面介绍的 get 用于捕获对属性的读取，set 用于捕获对属性的赋值。接下来我们将介绍各种各样的捕获器。

6.3.1 has捕获器

我们可以使用 handler.has 对 in 操作符隐藏任意属性。在前面 set 捕获器的示例中，虽然可以不对属性赋值，阻止读取带某种前缀的属性，但一些讨厌的使用者仍然可以通过 proxy 刺探出那些属性是否存在。以下是三种应对策略。

- 什么也不做，此时 key in proxy 会转换为 Reflect.has(target, key)，等价于 key in target。
- 无论 key 是否存在于 target 中，都返回 true 或 false 值。
- 抛出错误以表明此时 in 操作符是非法的。

抛出错误的做法太极端了，而且对只想隐藏某些属性的情况而言并没有帮助。这只会让人知道这些属性是受保护的。但抛出错误在一些情况下是可行的。比如，你想告诉使用者为什么操作失败，那么你就可以在错误消息中解释失败的原因。

最好的做法是通过返回 false 来告诉使用者，他们访问的属性并不在（in）对象中。此时最适合的做法就是返回表达式 key in target 的结果。

再回到 6.1.2 节中的示例。我们希望对带有前缀的属性的查询都返回 false，对其他属性的查询则采用默认行为。这样可以对那些不速之客隐藏不可访问的属性。

```
const handler = {
  get(target, key) {
    invariant(key, 'get')
    return Reflect.get(target, key)
  },
  set(target, key, value) {
    invariant(key, 'set')
```

```
    return Reflect.set(target, key, value)
  },
  has(target, key) {
    if (key.startsWith('_')) {
      return false
    }
    return Reflect.has(target, key)
  }
}
function invariant(key, action) {
  if (key.startsWith('_')) {
    throw new Error(`Can't ${ action } private "${ key }"
    property`)
  }
}
```

此时通过代理访问私有属性会返回 `false`，而使用者并不清楚这是什么情况。他们完全没有意识到我们是在有意隐藏信息。但 `_secret in target` 会返回 `true`，因为我们绕过了代理。这意味着我们照样可以通过严格控制访问规则而随意地操作底层对象，但使用者必须遵守规则。

```
const target = {
  _secret: 'securely-stored-value',
  wellKnown: 'publicly-known-value'
}
const proxy = new Proxy(target, handler)
console.log('wellKnown' in proxy)
// <- true
console.log('_secret' in proxy)
// <- false
console.log('_secret' in target)
// <- true
```

当然，也可以抛出异常。将访问私有空间当成不会导致无效状态的错误时，这是合适的。此时它与向第三方网站嵌入代码的安全问题性质不同。

注意，要想阻止 `Object#hasOwnProperty` 在私有空间找到属性，`has` 捕获器可帮不上忙。

```
console.log(proxy.hasOwnProperty('_secret'))
// <- true
```

6.4.1 节介绍的 `getOwnPropertyDescriptor` 捕获器也可以拦截 `Object#hasOwnProperty`，因此可以作为一个解决方案。

6.3.2 deleteProperty捕获器

将某个属性设置为 `undefined` 会清除它的值，但属性仍然还在。但像 `delete cat.furBall` 这样使用 `delete` 操作符则会将属性彻底从 `cat` 对象上删除。

```
const cat = { furBall: true }
cat.furBall = undefined
console.log('furBall' in cat)
// <- true
delete cat.furBall
console.log('furBall' in cat)
// <- false
```

前面阻止访问带有前缀的属性的示例存在问题：我们不能修改 _secret 属性的值，甚至无法通过 in 操作符获悉它是否存在，但还是可以通过 proxy 对象用 delete 操作符删除该属性！以下代码展示了这一点。

```
const target = { _secret: 'foo' }
const proxy = new Proxy(target, handler)
console.log('_secret' in proxy)
// <- false
console.log('_secret' in target)
// <- true
delete proxy._secret
console.log('_secret' in target)
// <- false
```

我们可以使用 handler.deleteProperty 来阻止删除操作。与 get 和 set 捕获器一样，在 deleteProperty 捕获器中抛出错误就可以阻止删除某个属性了。这时候抛出错误是可以接受的，因为我们希望使用者知道对带有前缀的属性的外部操作是禁止的。

```
const handler = {
  get(target, key) {
    invariant(key, 'get')
    return Reflect.get(target, key)
  },
  set(target, key, value) {
    invariant(key, 'set')
    return Reflect.set(target, key, value)
  },
  deleteProperty(target, key) {
    invariant(key, 'delete')
    return Reflect.deleteProperty(target, key)
  }
}
function invariant(key, action) {
  if (key.startsWith('_')) {
    throw new Error(`Can't ${ action } private "${ key }"
    property`)
  }
}
```

如果再像前面那样删除带有前缀的属性，比如从 proxy 删除 _secret，那么会导致异常。以下示例展示了加入 deleteProperty 捕获器后的效果。

```
const target = { _secret: 'foo' }
const proxy = new Proxy(target, handler)
console.log('_secret' in proxy)
// <- true
delete proxy._secret
// <- Error: Can't delete private "_secret" property
```

通过 proxy 访问 target 的使用者再也不能删除私有属性了。又少了一个麻烦!

6.3.3 defineProperty捕获器

ES5 引入的 Object.defineProperty 函数可以用属性 key 和描述符 descriptor 给 target 对象添加属性。Object.defineProperty(target, key, descriptor) 通常用于以下两种情况。

(1) 希望 getter 和 setter 得到跨浏览器的支持。
(2) 希望实现自定义的属性访问器。

手工添加的属性是可读的、可写的、可删的、可枚举的。

通过 Object.defineProperty 添加的属性默认是只读的、不可删除的、不可枚举的。此时的属性有点类似用 const 声明的绑定:虽然都是只读的,但并非不可变。

使用 Object.defineProperty 定义属性时,我们可以通过属性描述符自定义以下方面。

- configurable = false 禁用对属性描述符的大多数改变,同时使属性不可删除。
- enumerable = false 对 for..in 循环和 Object.keys 隐藏当前属性。
- writable = false 让属性只读。
- value = undefined 是当前属性的初始值。
- get = undefined 是作为属性 getter 的方法。
- set = undefined 是作为属性 setter 的方法,它接受一个新 value,然后更新属性的 value。

注意,你必须选择是配置 value 和 writable,还是配置 get 和 set。选择前者,你就是在配置一个数据描述符。像 pizza.topping = 'ham' 这样创建普通属性也会得到一个数据描述符。此时的 topping 有一个 value,它要么是 writable,要么不是 writable。选择后者,你就是在创建一个访问器描述符,它完全由用于该属性的 get 和 set(value) 的方法决定。

以下代码示例展示了两种属性描述符,这取决于定义属性采用的是声明方式,还是编程 API。这里使用了 Object.getOwnPropertyDescriptor,它接收传入的 target 对象和 key 属性,以取得我们所创建属性的描述符。

```
const pizza = {}
pizza.topping = 'ham'
Object.defineProperty(pizza, 'extraCheese', { value: true })
console.log(Object.getOwnPropertyDescriptor(pizza, 'topping'))
```

```
// {
//   value: 'ham',
//   writable: true,
//   enumerable: true,
//   configurable: true
// }
console.log(
  Object.getOwnPropertyDescriptor(pizza, 'extraCheese')
)
// {
//   value: true,
//   writable: false,
//   enumerable: false,
//   configurable: false
// }
```

handler.defineProperty 捕获器用于拦截要定义的属性。注意，这个捕获器既可以拦截声明式定义 pizza.extraCheese = false，也可以拦截 API 调用 Object.defineProperty(pizza, 'extraCheese', { value: false })。这个捕获器的第一个参数是 target 对象，第二个参数是属性 key，第三个参数是描述符 descriptor。

以下示例展示了如何阻止通过 proxy 给对象添加属性。如果处理器返回 false，属性的声明式定义在严格模式下会导致抛出异常，在非严格模式下则会静默失败。严格模式比非严格模式好，其严格的语义确保了优异的性能。因此，严格模式也是 ES6 模块的默认模式，第 8 章会对此进行介绍。为此，本书假设所有的代码示例都是在严格模式下运行的。

```
const handler = {
  defineProperty(target, key, descriptor) {
    return false
  }
}
const target = {}
const proxy = new Proxy(target, handler)
proxy.extraCheese = false
// <- TypeError: 'defineProperty' on proxy: trap returned false
```

再回到前缀属性的示例，我们也可以使用 defineProperty 捕获器来阻止通过代理创建私有属性。以下代码将重用 invariant 函数，以抛出异常的方式拒绝在私有空间中定义属性的行为。

```
const handler = {
  defineProperty(target, key, descriptor) {
    invariant(key, 'define')
    return Reflect.defineProperty(target, key, descriptor)
  }
}
function invariant(key, action) {
  if (key.startsWith('_')) {
    throw new Error(`Can't ${ action } private "${ key }"
```

```
    property`)
  }
}
```

接下来我们用一个 target 对象来试试。我们会尝试声明一个不带前缀的属性，再声明一个带前缀的属性。使用 proxy 在私有空间添加一个属性会导致抛出错误。

```
const target = {}
const proxy = new Proxy(target, handler)
proxy.topping = 'cheese'
proxy._secretIngredient = 'salsa'
// <- Error: Can't define private "_secretIngredient" property
```

proxy 对象通过一个捕获器成功地隐藏了 _secret 属性，该捕获器会通过 proxy[key] = value 和 Object.defineProperty(proxy, key, { value }) 隐藏这些属性。如果再将前面介绍的捕获器算进来，那我们已经阻止了对私有属性的读取、写入、查询和创建。

6.3.4 ownKeys捕获器

handler.ownKeys 方法可以返回一个属性数组，而这个数组也是 Reflect.ownKeys() 的结果。数组中应该包含 target 对象的所有属性：可枚举的、不可枚举的，以及符号属性。与之前一样，默认实现可以直接返回对被代理的 target 对象执行反射方法的结果。

```
const handler = {
  ownKeys(target) {
    return Reflect.ownKeys(target)
  }
}
```

这里拦截并不会影响 Object.keys 的输出，因为我们只是简单地将访问转发给了默认实现。

```
const target = {
  [Symbol('id')]: 'ba3dfcc0',
  _secret: 'sauce',
  _toppingCount: 3,
  toppings: ['cheese', 'tomato', 'bacon']
}
const proxy = new Proxy(target, handler)
for (const key of Object.keys(proxy)) {
  console.log(key)
  // <- '_secret'
  // <- '_toppingCount'
  // <- 'toppings'
}
```

注意，ownKeys 捕获器适用于以下所有操作。

- Reflect.ownKeys() 返回对象上所有自己的键。
- Object.getOwnPropertyNames() 只返回非符号属性。

- Object.getOwnPropertySymbols() 只返回符号属性。
- Object.keys() 只返回非符号可枚举属性。
- for..in 只返回非符号可枚举属性。

想要阻止访问带有前缀的属性时，我们可以先取得 Reflect.ownKeys(target) 的输出，然后从中过滤掉带有前缀的属性。这也正是 Object.getOwnPropertySymbols() 方法内部采取的策略。

以下示例对 Reflect.ownKeys(target) 的结果进行了过滤，只要不是字符串（那就是 Symbol 属性），则返回 true。然后过滤掉是字符串但以 _ 开头的属性。

```
const handler = {
  ownKeys(target) {
    return Reflect.ownKeys(target).filter(key => {
      const isStringKey = typeof key === 'string'
      if (isStringKey) {
        return !key.startsWith('_')
      }
      return true
    })
  }
}
```

如果在前面的代码中使用这里定义的 handler，那通过 proxy 只能取得非私有属性。注意，这里也没有返回 Symbol 属性，因为 Object.keys 在返回结果前过滤了 Symbol 属性。

```
const target = {
  [Symbol('id')]: 'ba3dfcc0',
  _secret: 'sauce',
  _toppingCount: 3,
  toppings: ['cheese', 'tomato', 'bacon']
}
const proxy = new Proxy(target, handler)
for (const key of Object.keys(proxy)) {
  console.log(key)
  // <- 'toppings'
}
```

这并不影响枚举 Symbol 属性，因为前面的 handler 并未过滤掉 Symbol 属性。Symbol 属性的类型是 symbol，因此 .filter 函数返回 true。

```
const target = {
  [Symbol('id')]: 'ba3dfcc0',
  _secret: 'sauce',
  _toppingCount: 3,
  toppings: ['cheese', 'tomato', 'bacon']
}
const proxy = new Proxy(target, handler)
for (const key of Object.getOwnPropertySymbols(proxy)) {
```

```
  console.log(key)
  // <- Symbol(id)
}
```

在符号及其他属性不受影响的情况下，我们成功地对枚举操作隐藏了带前缀“_”的属性。更重要的是，一个 ownKeys 捕获器就涵盖了所有的枚举操作，因此我们无须动用多个捕获器。唯一要注意的是，处理 Symbol 属性时必须小心一点。

6.4 高级代理捕获器

到目前为止，我们介绍的捕获器基本上都与属性访问和操作相关。除了接下来介绍的捕获器与属性访问相关，本章后面介绍的其他捕获器不再与属性相关，而是与要代理的对象本身相关。

6.4.1 getOwnPropertyDescriptor捕获器

在查询对象的某个属性描述符时，getOwnPropertyDescriptor 捕获器会触发。如果属性存在，那么它应该返回该属性的描述符，否则返回 undefined。当然，你也可以抛出异常，完全中断这个操作。

如果再以经典的私有属性为例，那么我们可以实现一个捕获器，以阻止使用者进一步取得私有属性的描述符，如下所示。

```
const handler = {
  getOwnPropertyDescriptor(target, key) {
    invariant(key, 'get property descriptor for')
    return Reflect.getOwnPropertyDescriptor(target, key)
  }
}
function invariant(key, action) {
  if (key.startsWith('_')) {
    throw new Error(`Can't ${ action } private "${ key }" property`)
  }
}
const target = {}
const proxy = new Proxy(target, handler)
Reflect.getOwnPropertyDescriptor(proxy, '_secret')
// <- Error: Can't get property descriptor for private
// "_secret" property
```

这样做的问题是，你其实已经告诉外部用户他们正在未授权的情况下访问带有前缀的属性。此时最好还是返回 undefined 来完全隐藏私有属性。这样私有属性的表现就与 target 对象确实没有的那些属性没什么区别了。以下示例表明，通过 Object.getOwnPropertyDescriptor 查询一个不存在的属性与查询 _secret 属性的结果相同。不在私有空间内的常规属性会返回常规的描述符。

```
const handler = {
  getOwnPropertyDescriptor(target, key) {
    if (key.startsWith('_')) {
      return
    }
    return Reflect.getOwnPropertyDescriptor(target, key)
  }
}
const target = {
  _secret: 'sauce',
  topping: 'mozzarella'
}
const proxy = new Proxy(target, handler)
console.log(Object.getOwnPropertyDescriptor(proxy, 'dressing'))
// <- undefined
console.log(Object.getOwnPropertyDescriptor(proxy, '_secret'))
// <- undefined
console.log(Object.getOwnPropertyDescriptor(proxy, 'topping'))
// {
//   value: 'mozzarella',
//   writable: true,
//   enumerable: true,
//   configurable: true
// }
```

`getOwnPropertyDescriptor` 捕获器可以拦截 `Object#hasOwnProperty` 的实现，后者依赖属性描述符来确认属性是否存在。

```
console.log(proxy.hasOwnProperty('topping'))
// <- true
console.log(proxy.hasOwnProperty('_secret'))
// <- false
```

在隐藏信息时，最好能将它们归到一个不同于原先类别的其他类别，这样既可以隐藏它们的踪迹，又不会影响其他操作。如果为隐藏信息而抛出错误，那就会引人怀疑：为什么它抛出错误，而不是返回 `undefined`？这个属性一定存在，只是不允许访问。这与 HTTP API 设计中对敏感资源返回“404 Not Found”没什么不同。比如，当无权用户访问管理后台时，尽管技术上正确，但不要返回“401 Unauthorized”。

如果排错的优先级高于安全考量，则可以考虑使用 `throw`。不管怎样，只有充分理解使用场景才能找出最优及干扰最少的方案。

6.4.2 apply捕获器

`apply` 捕获器很有意思，它是专门为函数量身定制的。当代理的 `target` 函数被调用时，`apply` 捕获器就会触发。以下代码示例中的所有语句都会通过代理的 `handler` 对象的 `apply` 捕获器。

```
proxy('cats', 'dogs')
proxy(...['cats', 'dogs'])
```

```
proxy.call(null, 'cats', 'dogs')
proxy.apply(null, ['cats', 'dogs'])
Reflect.apply(proxy, null, ['cat', 'dogs'])
```

apply 捕获器接受三个参数：

- target 是被代理的函数；
- ctx 是函数被调用时作为 target 中 this 值的上下文对象；
- args 是一个函数被调用时传给 target 的数组。

不修改输出的默认实现会返回调用 Reflect.apply 的结果。

```
const handler = {
  apply(target, ctx, args) {
    return Reflect.apply(target, ctx, args)
  }
}
```

除了能够打印每次通过 proxy 调用函数时的参数，这个捕获器还可以用于添加额外的参数或修改函数调用的结果。所有这些功能都可以在不改变底层 target 函数的前提下实现，因此这个捕获器可以重用于需要同样额外功能的任何函数。

以下示例代理 sum 函数时应用了 twice 处理器，它通过 apply 捕获器将每次求和的结果都乘以 2。如果不直接调用 sum，而是调用 proxy，那么结果就总能翻倍，而且不影响原先的代码。

```
const twice = {
  apply(target, ctx, args) {
    return Reflect.apply(target, ctx, args) * 2
  }
}
function sum(a, b) {
  return a + b
}
const proxy = new Proxy(sum, twice)
console.log(proxy(1, 2))
// <- 6
```

我们再来看看另一种情况。假设我们想要在调用不同函数时都保持上下文 this 不变。以下示例定义了一个 logger 对象，它有一个 test 方法，可以返回 logger 对象自己。

```
const logger = {
  test() {
    return this
  }
}
```

要想每次调用 test 方法都返回 logger，那么我们可以将 test 与 logger 绑定起来，如下所示。

```
logger.test = logger.test.bind(logger)
```

问题是，对于 logger 对象中的每个函数，如果想要确保函数内部的 this 指向 logger，那么我们必须这样绑定一次。另一种办法是借助一个自定义 get 捕获器的代理，如果返回的是函数，则将该函数绑定到 target 对象。

```
const selfish = {
  get(target, key) {
    const value = Reflect.get(target, key)
    if (typeof value !== 'function') {
      return value
    }
    return value.bind(target)
  }
}
const proxy = new Proxy(logger, selfish)
```

这个实现对任何对象（包括类）都适用，而且不必做什么修改。以下代码展示了原来的 logger 很容易被 .call 攻击，从而改变 this 的指向。但 proxy 对象面对同样的攻击则完全不受影响。

```
const something = {}
console.log(logger.test() === logger)
// <- true
console.log(logger.test.call(something) === something)
// <- true
console.log(proxy.test() === logger)
// <- true
console.log(proxy.test.call(something) === logger)
// <- true
```

不过，像这样借助 selfish 保持上下文也有一个小问题，即每次通过 proxy 获得函数的引用时，都是调用 value.bind(target) 返回的一个全新的绑定版函数。相应地，函数与函数就不相等了。这可能会导致一些令人困惑的行为，如下所示。

```
console.log(proxy.test !== proxy.test)
// <- true
```

这个问题可以通过 WeakMap 解决。我们可以将 selfish 改写成一个工厂函数。在这个函数体内，我们通过 cache 保留一份已经绑定方法的缓存，这样每个函数就只会创建一个绑定版。同时，我们还让 selfish 函数接收一个想要被代理的 target 对象。如此一来，将方法绑定到哪个对象就是实现要考虑的问题了。

```
function selfish(target) {
  const cache = new WeakMap()
  const handler = {
    get(target, key) {
      const value = Reflect.get(target, key)
      if (typeof value !== 'function') {
```

```
        return value
      }
      if (!cache.has(value)) {
        cache.set(value, value.bind(target))
      }
      return cache.get(value)
    }
  }
  const proxy = new Proxy(target, handler)
  return proxy
}
```

现在，我们可以缓存绑定的函数并通过原始值来跟踪它们了，而且每次都能返回相同的对象。再进行以下这样的简单比较就不会令人惊讶了。

```
const selfishLogger = selfish(logger)
console.log(selfishLogger.test === selfishLogger.test)
// <- true
console.log(selfishLogger.test() === selfishLogger)
// <- true
console.log(selfishLogger.test.call(something) ===
   selfishLogger)
// <- true
```

当我们想让对象的所有方法都绑定到宿主对象时，selfish 函数可以重用于任何对象。这对类非常有用，因为类非常依赖于 this 指向实例对象。

将方法绑定到父对象的办法非常多，各有优缺点。使用代理可能是其中最方便的、麻烦最少的，只是浏览器的支持还不够好，而且 Proxy 的实现比较慢。

我们并没有在 selfish 的例子中用到 apply 捕获器，这表明并不是所有东西都能通用。如果在这种情况下使用 apply 捕获器，那么 selfish 就要返回 value 函数的代理，然后在 value 代理的 apply 捕获器中再返回一个绑定的函数。这听起来似乎更正确，因为没有用 .bind 而使用了 Reflect.apply，但我们还是需要 WeakMap 缓存和 selfish 代理。从关注点分离和可维护的角度来说，我们需要额外添加一层抽象，还要使用第二个代理，然后得到一个值。考虑到两个代理层之间一定会存在某种程度的耦合，最好还是将所有逻辑都封装在一个抽象层中。抽象虽好，但过多的抽象可能比原来要解决的问题还要难以处理。

在类的构造器中使用 .bind 语句实现抽象时，怎样才算合理？此类问题通常没有唯一的答案，视具体情况而定。但在设计组件系统时，这些是必须要考虑的。增加抽象层可以避免重复自己，从而避免为解决复杂的问题而将事情复杂化。

6.4.3 construct捕获器

construct 捕获器拦截的是 new 操作符的调用。以下代码实现了一个自定义的 construct 捕获器，其行为与 construct 捕获器一致。这里用到了扩展操作符和 new 关键字，因此我们

可以向 Target 构造函数传入任何参数。

```
const handler = {
  construct(Target, args) {
    return new Target(...args)
  }
}
```

前面的示例与使用 Reflect.construct 效果相同，如下所示。但是这里没有使用扩展操作符。反射方法镜像了代理捕获器的方法签名，因此 Reflect.construct 的签名同样也包含 Target，args，与 construct 捕获器方法相同。

```
const handler = {
  construct(Target, args) {
    return Reflect.construct(Target, args)
  }
}
```

使用 construct 捕获器不需要工厂函数即可修改或扩展对象的行为，甚至改变其实现。但注意，代理的目的必须要明确，而且这个目的不能过于扰乱底层对象的实现。具体来说，将 construct 代理捕获器当作开关来切换不同的底层类就不是好的抽象思路，因为一个简单的函数就够用了。

construct 捕获器最常见的用例是重组构造器的参数或做一些本应由构造器来做的事，比如记录日志或记录创建的对象实例。

以下示例展示了在不改变类实现的情况下，如何通过代理向部分使用者提供不同的体验。在使用 ProxiedTarget 时，我们可以用构造器参数在目标实例上声明一个 name 属性。

```
const handler = {
  construct(Target, args) {
    const [ name ] = args
    const target = Reflect.construct(Target, args)
    target.name = name
    return target
  }
}
class Target {
  hello() {
    console.log(`Hello, ${ this.name }!`)
  }
}
```

对这个例子而言，我们可以直接修改 Target，让它在构造器中接收一个 name 参数，并将其保存为实例属性。但现实往往不允许我们这么做。由于种种原因，我们可能无法直接修改这个类，比如代码不是我们编写的，或者已经有别的代码依赖于当前的对象结构。以下代码展示了使用 Target 的实际情形，既有其自己的 API，也有基于 construct 捕获器修改的 ProxiedTarget API。

```
const target = new Target()
target.name = 'Nicolás'
target.hello()
// <- 'Hello, Nicolás'

const ProxiedTarget = new Proxy(Target, handler)
const proxy = new ProxiedTarget('Nicolás')
proxy.hello()
// <- 'Hello, Nicolás'
```

注意，箭头函数不能用作构造器，否则不能对这个类使用 `construct` 捕获器。接下来我们再看看剩下的几个捕获器。

6.4.4 getPrototypeOf捕获器

我们可以使用 `handler.getPrototypeOf` 方法作为捕获器来拦截以下所有操作。

- 访问 `Object#__proto__` 属性。
- 调用 `Object#isPrototypeOf` 方法。
- 调用 `Object#getPrototypeOf` 方法。
- 调用 `Reflect.getPrototypeOf` 方法。
- 使用 `instanceof` 操作符。

这个捕获器很强大，因为我们可以通过它动态地指定要报告的底层对象的原型。

比如，我们可以使用这个捕获器让某个对象在通过代理访问时被当成 `Array`。以下代码就实现了这样一个代理，它返回 `Array.prototype` 作为被代理对象的原型。注意，测试代理对象是否为 `Array` 的实例时，`instanceof` 操作符确实返回了 `true`。

```
const handler = {
  getPrototypeOf: target => Array.prototype
}
const target = {}
const proxy = new Proxy(target, handler)
console.log(proxy instanceof Array)
// <- true
```

实际上，代理对象本身还不是数组。以下代码表明，尽管测试显示代理对象是一个数组，但这个对象上并没有 `Array#push` 方法。

```
console.log(proxy.push)
// <- undefined
```

当然，我们可以继续“装扮”代理对象，让它最终拥有数组的行为。为此，我们需要使用 `get` 捕获器让 `Array.prototype` 成为实际后端对象 `target` 的替补。如果哪个属性无法在 `target` 上找到，则可以再次通过反射到 `Array.prototype` 上查找。事实证明，可以在对象

上使用数组的方法。

```
const handler = {
  getPrototypeOf: target => Array.prototype,
  get(target, key) {
    return (
      Reflect.get(target, key) ||
      Reflect.get(Array.prototype, key)
    )
  }
}
const target = {}
const proxy = new Proxy(target, handler)
```

注意，此时的 proxy.push 指向的是 Array#push 方法。我们可以像使用数组那样不动声色地使用代理对象，同时打印出来的代理对象显示它还是对象，并不是数组 ['first', 'second']。

```
console.log(proxy.push)
// <- function push() { [native code] }
proxy.push('first', 'second')
console.log(proxy)
// <- { 0: 'first', 1: 'second', length: 2 }
```

与 getPrototypeOf 捕获器相对的是 setPrototypeOf。

6.4.5 setPrototypeOf捕获器

ES6 有一个 Object.setPrototypeOf 方法，用于将一个对象的原型设置成另一个对象。与使用特殊属性 __proto__ 相比，这是改变对象原型的首选方式，虽然多数浏览器都支持前者，但 ES6 已将其废弃。

废弃的意思是浏览器不再认可使用 __proto__。如果是不同的运行时，这可能意味着这个属性将来会被删除。但 Web 平台不会出现向后不兼容的情况，因此 __proto__ 属性不会消失。尽管如此，废弃的属性终究是废弃的属性，最好别再使用。记住，如果以后需要修改对象的原型，尽量使用 Object.setPrototypeOf，不要使用 __proto__。

我们可以用 handler.setPrototypeOf 来设置 Object.setPrototypeOf 的捕获器。以下代码并没有改变将原型修改为 base 的默认行为。注意，还有一个与这里的 Object.setPrototypeOf 等价的反射方法：Reflect.setPrototypeOf。

```
const handler = {
  setPrototypeOf(target, proto) {
    Object.setPrototypeOf(target, proto)
  }
}
const base = {}
```

```
function Target() {}
const proxy = new Proxy(Target, handler)
proxy.setPrototypeOf(proxy, base)
console.log(proxy.prototype === base)
// <- true
```

setPrototypeOf 捕获器也有一些使用场景。比如，我们可以编写一个空方法体，然后通过捕获器“蒸发”对 Object.setPrototypeOf 的调用，也就是什么都不做。如果你认为新原型有问题或者就想阻止用户修改被代理对象的原型，也可以通过抛出错误明确地告知用户操作失败。

你可以像以下这样实现捕获器。如果这个代理被传给第三方代码，那么限制对底层 Target 的访问可以减少一些安全威胁。这样一来，代理的使用者就无法修改底层对象的原型。

```
const handler = {
  setPrototypeOf(target, proto) {
    throw new Error('Changing the prototype is forbidden')
  }
}
const base = {}
function Target() {}
const proxy = new Proxy(Target, handler)
proxy.setPrototypeOf(proxy, base)
// <- Error: Changing the prototype is forbidden
```

此时最好抛出一个提示操作失败的异常，让用户知道发生了什么。显式禁止修改原型后，用户才会考虑其他选择。如果不抛出异常，用户最终可能也会通过调试发现原来不允许修改原型。与其眼睁睁看着用户浪费时间，不如直截了当地告诉他们原因。

6.4.6 preventExtensions捕获器

我们可以用 handler.preventExtensions 来捕获 ES5 新增的 Object.preventExtensions 方法。被阻止扩展后，就不能给对象新增属性了，因为不允许扩展对象。

想象一下，你希望可以选择性地阻止对一些对象的扩展，而不是全部阻止。此时可以用 WeakSet 来保存可扩展对象。如果对象在这个集合中，那么 preventExtensions 可以捕获相应的请求，然后就地丢弃它们。

以下代码实现了上述场景：将可扩展对象保存在一个 WeakSet 中，并阻止其他对象被扩展。

```
const canExtend = new WeakSet()
const handler = {
  preventExtensions(target) {
    const canPrevent = !canExtend.has(target)
    if (canPrevent) {
      Object.preventExtensions(target)
    }
```

```
      return Reflect.preventExtensions(target)
    }
  }
```

有了 WeakSet 和 handler 后，我们就可以创建一个目标对象及其代理，然后将目标对象添加到集合中。接着尝试在代理上执行 Object.preventExtensions，我们会发现阻止对 target 进行扩展并未成功。这正是我们想要的，因为 target 位于 canExtend 集合。需要注意的是，虽然我们在这里看到了一个 TypeError 异常（因为使用者试图阻止扩展的操作被捕获到并丢弃了），但非严格模式下只会静默失败。

```
const target = {}
const proxy = new Proxy(target, handler)
canExtend.add(target)
Object.preventExtensions(proxy)
// 捕获器返回false
// trap returned falsy
```

如果先将 target 从 canExtend 集合中删掉，再调用 Object.preventExtensions，那么 target 就会像最初那样变得不能扩展了。以下代码展示了这个过程。

```
const target = {}
const proxy = new Proxy(target, handler)
canExtend.add(target)
canExtend.delete(target)
Object.preventExtensions(proxy)
console.log(Object.isExtensible(proxy))
// <- false
```

6.4.7 isExtensible捕获器

可扩展对象就是可以为其添加属性的对象。换句话说，你可以扩展这个对象。

handler.isExtensible 方法可以用于记录或监督对 Object.isExtensible 的调用，但不能左右对象是否可以扩展的事实。这是因为这个捕获器被施加了严格的控制：如果 Object.isExtensible(proxy) !== Object.isExtensible(target)，则抛出 TypeError。我们能通过这个捕获器做的事情极其有限。

虽然这个捕获器除了监督记录几乎没什么用，但如果不想让用户知道底层对象是否可扩展，我们还可以抛出错误。

正如前面所说，代理的用途其实非常广泛。以下列举了一些可以用代理完成的事情，但这些只是冰山一角。

- 给简单 JavaScript 对象添加验证规则并强制验证。
- 监控通过代理对底层对象执行的每一次操作。

- 实现你自己的可观察对象。
- 在不改变底层实现的前提下装饰和扩展对象。
- 让对象的某些属性对外部用户完全不可见。
- 只要你认为用户不应该再访问某个对象，随时可以撤销其访问权。
- 修改传递给被代理方法的参数。
- 修改被代理方法返回的结果。
- 通过代理阻止删除某些属性。
- 根据期望的属性描述符阻止定义新的属性。
- 调控类构造器中的参数。
- 返回非 `new` 及构造器创建的结果。
- 给对象换个原型。

代理是 ES6 中极其强大的新特性，有着极其广泛的应用，它们本身也是精心规划、反复推敲的结果。就当前的 JavaScript 引擎而言，与代理相关的操作的性能还不够理想，而且几乎不可能优化。因此，对于追求极致性能的应用来说，代理会显得有点力不从心。

与此同时，试图做太多事情的复杂代理也会让使用者感到困惑。我们的建议是，一般情况下不必考虑使用代理，除非有简明可循的使用规则。如果使用代理，那么一定要确保副作用不会太多，否则即使有文档，也可能将问题复杂化。

第 7 章

ES6中内置API的改进

至此，我们讨论了所有的新语法，比如对象属性值简写、箭头函数、解构及生成器。我们还讨论了所有的新内置 API，如 `WeakMap`、`Proxy` 及 `Symbol`。本章将聚焦于那些在 ES6 问世前就已经存在，且在 ES6 中被加以改进的内置 API。这些改进主要包括大部分的新实例方法、属性以及工具方法。

7.1　数字

ES6 引入了二进制和八进制的数字字面量表示。

7.1.1　二进制和八进制字面量

在 ES6 问世前，当涉及解析二进制整数时，最好的做法是将它们传入 `parseInt` 中，并带上基数 `2`。

```
parseInt('101', 2)
// <- 5
```

现在你可以使用一个新的前缀 `0b` 来表示二进制整数，你也可以使用前缀 `0B`，其中 `B` 大写。这两种形式是等价的。

```
console.log(0b000) // <- 0
console.log(0b001) // <- 1
console.log(0b010) // <- 2
console.log(0b011) // <- 3
console.log(0b100) // <- 4
```

```
console.log(0b101) // <- 5
console.log(0b110) // <- 6
console.log(0b111) // <- 7
```

在 ES3 中，`parseInt` 将以 `0` 开头的数字字符串视为一个八进制数值。然而，一旦你忘记指定基数为 `10`，事情就会变得奇怪起来。因此，为了避免 `012` 这样的输入被不小心解析为 `10`，指定 `10` 为基数成为了最佳实践。

```
console.log(parseInt('01'))
// <- 1
console.log(parseInt('012'))
// <- 10
console.log(parseInt('012', 10))
// <- 12
```

当 ES5 的时代来临，`parseInt` 的默认基数从 `8` 变成了 `10`。但为了向后兼容，官方依旧建议指定一个基数。要想将一个字符串解析为八进制数值，就需要明确地传入一个基数 `8` 作为第二个参数。

```
console.log(parseInt('100', 8))
// <- 64
```

在 ES6 的新规范中，你可以使用 `0o` 前缀来表示八进制数值。你也可以使用 `0O` 前缀，两种形式是等价的。然而，`0` 加上大写的 `O` 在某些字体中很难区分，这就是官方建议使用小写 `0o` 的原因。

```
console.log(0o001) // <- 1
console.log(0o010) // <- 8
console.log(0o100) // <- 64
```

在其他语言中，你可能习惯于用 `0x` 前缀来表示十六进制数值。这样的写法早在 ES5 就引入 JavaScript 了。这种十六进制数字的前缀可以写成 `0x` 或者 `0X`，如下所示。

```
console.log(0x0ff) // <- 255
console.log(0xf00) // <- 3840
```

除了以上介绍的关于二进制和八进制语法的少量变动外，ES6 还为 `Number` 对象增添了一些其他方法。接下来我们将探讨 `Number` 对象的前四个方法：`Number.isNaN`、`Number.isFinite`、`Number.parseInt` 和 `Number.parseFloat`。它们已经作为函数存在于全局命名空间中。然而，`Number` 对象中的这些方法有些许不同，即它们不会在产生结果前强制地将非数字值转换为数字。

7.1.2 Number.isNaN

这个方法和全局的 `isNaN` 几乎相同。不同点是，`Number.isNaN` 的返回值是判断传入的 `value` 是否为 `NaN` 的结果，而 `isNaN` 的返回值是判断传入的 `value` 是否为非数字的结果。这两个

问题的答案稍有不同。

以下代码表明，任何传入 Number.isNaN 的非 NaN 值都会返回 false，只有传入 NaN 时才会返回 true。值得注意的是，在最后一个示例中，我们实际上是向 Number.isNaN 传入了一个 NaN，因为这个传入的值是两个字符串相除的结果。

```
Number.isNaN(123)
// <- false，整型数字不是NaN
Number.isNaN(Infinity)
// <- false，Infinity不是NaN
Number.isNaN('a hundred')
// <- false，'a hundred'不是NaN
Number.isNaN(NaN)
// <- true，NaN是NaN
Number.isNaN('a hundred' / 'two')
// <- true，'a hundred' / 'two'是NaN，NaN是NaN
```

相反，与 NaN 比较前，isNaN 方法就对传入的值进行了类型转换。结果显而易见，返回值是不同的。在以下示例中，因为 isNaN 一开始就将 value 转换成了 Number 类型，所以与使用 Number.isNaN 的结果就大不一样了。

```
isNaN('a hundred')
// <- true，因为Number('a hundred')值为NaN
isNaN(new Date())
// <- false，因为Number(new Date())使用Date#valueOf，
// 返回值是一个Unix时间戳
```

Number.isNaN 要比其全局版本更为精确，因为它不包含类型转换。然而，一些场景仍然会使 Number.isNaN 变成烦恼的根源。

首先，isNaN 在比较开始前就进行了类型转换，但 Number.isNaN 不是。因此，isNaN 和 Number.isNaN 都没有回答"这个值难道不是一个数字吗"这个问题，而是回答了"一个值或一个数字是否为 NaN"的问题。

在大多数使用场景中，你实际上是想知道一个值是否可以被视为数字，即 typeof NaN === number，或者说这个值是不是数字。以下代码示例中的 isNumber 函数正好回答了这个问题。注意，这个函数同时使用了 isNaN 和 Number.isNaN 来进行类型检查。typeof 可以区分出一个值是否为数字类型，除非这个值是 NaN，因此我们过滤掉了那些假阳性的结果。

```
function isNumber(value) {
  return typeof value === 'number' && !Number.isNaN(value)
}
```

你可以用以上方法来区分一个值是否为数字，以下示例展现了 isNumber 是如何运作的。

```
isNumber(1)
// <- true
isNumber(Infinity)
```

```
// <- true
isNumber(NaN)
// <- false
isNumber('two')
// <- false
isNumber(new Date())
// <- false
```

有这么一个函数，它已经存在于这门语言中，并且从某些程度来说有点像我们定义的 `isNumber` 函数，它就是 `isFinite`。

7.1.3 Number.isFinite

`isFinite` 少有人知，但其实早就存在于 ES3 中。它返回一个布尔值以表明提供的 `value` 是否不匹配 `Infinity`、`-Infinity` 和 `NaN` 中的任何一个。

`isFinite` 方法会通过 `Number(value)` 对值进行强制转换，而 `Number.isFinite` 则不会。这意味着那些可以强制转换成非 `NaN` 数字的值将被 `isNumber` 视为有限数字，即使它们不是显式数字。

以下是一些使用全局 `isFinite` 函数的示例。

```
isFinite(NaN)
// <- false
isFinite(Infinity)
// <- false
isFinite(-Infinity)
// <- false
isFinite(null)
// <- true，因为Number(null)值为0
isFinite(-13)
// <- true，因为Number(-13)值为-13
isFinite('10')
// <- true，因为Number('10')值为10
```

使用 `Number.isFinite` 更安全，因为它不会出现意想不到的转换。要想将 `value` 强制转换为数字表示形式，你可以随时使用 `Number.isFinite(Number(value))`。分离转换与计算会使代码更明确。

以下是使用 `Number.isFinite` 方法的几个示例。

```
Number.isFinite(NaN)
// <- false
Number.isFinite(Infinity)
// <- false
Number.isFinite(-Infinity)
// <- false
Number.isFinite(null)
// <- false，因为null不是一个数字
```

```
Number.isFinite(-13)
// <- true
Number.isFinite('10')
// <- false，因为'10'不是一个数字
```

为 `Number.isFinite` 创建 ponyfill 将会为非数字的值返回 `false`，这可以有效地关闭类型转换功能，然后对输入值调用 `isFinite`。

```
function numberIsFinite(value) {
  return typeof value === 'number' && isFinite(value)
}
```

7.1.4 Number.parseInt

`Number.parseInt` 方法的工作原理与 `parseInt` 相同，事实上，它们就是相同的。

```
console.log(Number.parseInt === parseInt)
// <- true
```

`parseInt` 函数支持字符串中的十六进制字面量符号，甚至没有必要指定 `radix`：基于 `0x` 前缀，`parseInt` 可以推断该数字为十六进制。

```
parseInt('0xf00')
// <- 3840
parseInt('0xf00', 16)
// <- 3840
```

如果提供了另一个 `radix`，则 `parseInt` 会在第一个非数字字符后跳出。

```
parseInt('0xf00', 10)
// <- 0
parseInt('5xf00', 10)
// <- 5，可见这里没有特殊处理
```

虽然 `parseInt` 接受十六进制字面量符号字符串形式的输入，但其接口在 ES6 中并没有改变。因此，二进制和八进制字面量符号字符串将不会这样解析。ES6 引入了一种新的不一致性：`parseInt` 可以解析 `0x`，但不能解析 `0b` 和 `0o`。

```
parseInt('0b011')
// <- 0
parseInt('0b011', 2)
// <- 0
parseInt('0o100')
// <- 0
parseInt('0o100', 8)
// <- 0
```

要想用 `parseInt` 来解析这些字面量，你可以在 `parseInt` 前先删除它们的前缀。你还需要指定相应二进制或八进制的 `radix` 值。

```
parseInt('0b011'.slice(2), 2)
// <- 3
parseInt('0o110'.slice(2), 8)
// <- 72
```

相反，Number 函数完全可以将这些字符串转换为正确的数字。

```
Number('0b011')
// <- 3
Number('0o110')
// <- 72
```

7.1.5 Number.parseFloat

与 parseInt 一样，parseFloat 没有做任何修改就添加到 Number 了。

```
console.log(Number.parseFloat === parseFloat)
// <- true
```

好在 parseFloat 不会对十六进制字符串进行任何特殊转换，这意味着 Number.parseFloat 不太可能引起混淆。

出于完整性的目的，parseFloat 函数才添加到 Number 中。在未来的语言版本中，全局命名空间污染将会减少。用于特定目的时，函数会添加到相关的内置中，而不是全局。

7.1.6 Number.isInteger

Number.isInteger 是 ES6 中的一种新方法，它以前不能用作全局函数。如果提供的 value 是没有小数部分的有限数字，则 isInteger 方法返回 true。

```
console.log(Number.isInteger(Infinity)) // <- false
console.log(Number.isInteger(-Infinity)) // <- false
console.log(Number.isInteger(NaN)) // <- false
console.log(Number.isInteger(null)) // <- false
console.log(Number.isInteger(0)) // <- true
console.log(Number.isInteger(-10)) // <- true
console.log(Number.isInteger(10.3)) // <- false
```

你可能想将以下代码段当作 Number.isInteger 的 ponyfill 来使用。模数运算返回的是除以相同操作数的余数。如果除以一，我们可以得到小数部分。如果结果为 0，那就意味着这个数字是一个整数。

```
function numberIsInteger(value) {
  return Number.isFinite(value) && value % 1 === 0
}
```

接下来我们将深入探讨浮点运算，其中有很多有趣的案例。

7.1.7　Number.EPSILON

EPSILON 属性是 Number 内置中新添加的常数值。以下代码段展示了它的值。

```
Number.EPSILON
// <- 2.220446049250313e-16
Number.EPSILON.toFixed(20)
// <- '0.00000000000000022204'
```

我们来看看浮点运算的典型示例。

```
0.1 + 0.2
// <- 0.30000000000000004
0.1 + 0.2 === 0.3
// <- false
```

这个操作的误差是多少？我们可以移动操作数并找出答案。

```
0.1 + 0.2 - 0.3
// <- 5.551115123125783e-17
5.551115123125783e-17.toFixed(20)
// <- '0.00000000000000005551'
```

可以用 Number.EPSILON 来判断这个误差是否小到可以忽略不计；Number.EPSILON 表示浮点运算舍入操作的安全误差范围。

```
5.551115123125783e-17 < Number.EPSILON
// <- true
```

以下代码可以用来确定浮点运算的结果是否在预期的误差范围内。我们使用 Math.abs，这样 left 和 right 的位置就无关紧要了。换句话说，withinMarginOfError(left, right) 会产生与 withinMarginOfError(right, left) 相同的结果。

```
function withinMarginOfError(left, right) {
  return Math.abs(left - right) < Number.EPSILON
}
```

以下代码段展示了 withinMarginOfError 的实际使用情况。

```
withinMarginOfError(0.1 + 0.2, 0.3)
// <- true
withinMarginOfError(0.2 + 0.2, 0.3)
// <- false
```

在使用浮点表示法时，并不是每个整数都可以被精确表示。

7.1.8　Number.MAX_SAFE_INTEGER和Number.MIN_SAFE_INTEGER

Number.MAX_SAFE_INTEGER 是 JavaScript 中可以安全且精确表示的最大整数，或者是 IEEE

754 标准[1] 规定的能用浮点表示整数的任何语言可以表示的最大整数。以下代码展示了 Number.MAX_SAFE_INTEGER 有多大。

```
Number.MAX_SAFE_INTEGER === Math.pow(2, 53) - 1
// <- true
Number.MAX_SAFE_INTEGER === 9007199254740991
// <- true
```

正如你所预料的那样，最大值也有相反的常数——最小值，即 Number.MAX_SAFE_INTEGER 对应的负数。

```
Number.MIN_SAFE_INTEGER === -Number.MAX_SAFE_INTEGER
// <- true
Number.MIN_SAFE_INTEGER === -9007199254740991
// <- true
```

浮点运算在 [MIN_SAFE_INTEGER, MAX_SAFE_INTEGER] 范围之外变得不可靠。1 === 2 语句的结果是 false，因为它们是不同的值。但如果我们给每个数加上 Number.MAX_SAFE_INTEGER，那么 1 === 2 似乎是正确的。

```
1 === 2
// <- false
Number.MAX_SAFE_INTEGER + 1 === Number.MAX_SAFE_INTEGER + 2
// <- true
Number.MIN_SAFE_INTEGER - 1 === Number.MIN_SAFE_INTEGER - 2
// <- true
```

在检查一个整数是否安全时，Number.isSafeInteger 函数已经被添加到该语言中。

7.1.9 Number.isSafeInteger

对于 [MIN_SAFE_INTEGER, MAX_SAFE_INTEGER] 范围内的任何整数，此方法都会返回 true。与 ES6 引入的其他 Number 方法一样，该方法不涉及任何类型的强制转换。输入必须是整型数字，并且上述范围才能使此方法返回 true。以下代码展示了一组全面的输入和输出集。

```
Number.isSafeInteger('one') // <- false
Number.isSafeInteger('0') // <- false
Number.isSafeInteger(null) // <- false
Number.isSafeInteger(NaN) // <- false
Number.isSafeInteger(Infinity) // <- false
Number.isSafeInteger(-Infinity) // <- false
Number.isSafeInteger(Number.MIN_SAFE_INTEGER - 1) // <- false
Number.isSafeInteger(Number.MIN_SAFE_INTEGER) // <- true
Number.isSafeInteger(1) // <- true
Number.isSafeInteger(1.2) // <- false
Number.isSafeInteger(Number.MAX_SAFE_INTEGER) // <- true
Number.isSafeInteger(Number.MAX_SAFE_INTEGER + 1) // <- false
```

注 1：IEEE 754 是浮点标准。

要想验证一个操作的结果是否在界限内，我们不仅要验证结果，还要验证这两个操作数[2]。操作数之一或者二者都可能超出范围，而结果在范围内但并不正确。同样，即使两个操作数在范围内，结果也可能超出范围。检查所有的 `left`、`right` 和 `left op right` 的结果是很有必要的，这样我们才可以信任该结果。

在以下示例中，两个操作数都在范围内，但结果不正确。

```
Number.isSafeInteger(9007199254740000)
// <- true
Number.isSafeInteger(993)
// <- true
Number.isSafeInteger(9007199254740000 + 993)
// <- false
9007199254740000 + 993
// <- 9007199254740992, should be 9007199254740993
```

即使操作数超出范围，某些操作和数字还是可能会返回正确的结果，如下所示。然而，不能确保正确结果的事实意味着这些操作是不可信的。

```
9007199254740000 + 994
// <- 9007199254740994
```

以下示例中的其中一个操作数超出了范围，因此我们不能相信结果是准确的。

```
Number.isSafeInteger(9007199254740993)
// <- false
Number.isSafeInteger(990)
// <- true
Number.isSafeInteger(9007199254740993 + 990)
// <- false
9007199254740993 + 990
// <-  9007199254741982，应为9007199254741983
```

最后一个示例中的减法会产生一个范围内的结果，但这个结果也是不准确的。

```
Number.isSafeInteger(9007199254740993)
// <- false
Number.isSafeInteger(990)
// <- true
Number.isSafeInteger(9007199254740993 - 990)
// <- true
9007199254740993 - 990
// <-  9007199254740002，应为9007199254740003
```

如果两个操作数都超出了范围，即使结果不正确，输出也可能会在安全范围内。

```
Number.isSafeInteger(9007199254740995)
// <- false
```

注 2：Axel Rauschmayer 博士在 *New Number and Math Features in ES6* 一文中指出了这一点。

```
Number.isSafeInteger(9007199254740993)
// <- false
Number.isSafeInteger(9007199254740995 - 9007199254740993)
// <- true
9007199254740995 - 9007199254740993
// <- 4，应为2
```

我们得出的结论是，判断操作是否产生正确输出的唯一安全方法是使用如下所示的效用函数。如果无法确定其中的一个或两个操作数是否在范围内，那么结果可能不准确，这是一个问题。在这些情况下，最好能够抛出异常，并纠正错误，但这是针对你的程序。实际上，抓住这些难以处理的错误才是重要的部分。

```
function safeOp(result, ...operands) {
  const values = [result, ...operands]
  if (!values.every(Number.isSafeInteger)) {
    throw new RangeError('Operation cannot be trusted!')
  }
  return result
}
```

你可以使用 `safeOp` 来确保所有操作数（包括 `result`）都在安全范围内。

```
safeOp(9007199254740000 + 993, 9007199254740000, 993)
// <- RangeError: Operation cannot be trusted!
safeOp(9007199254740993 + 990, 9007199254740993, 990)
// <- RangeError: Operation cannot be trusted!
safeOp(9007199254740993 - 990, 9007199254740993, 990)
// <- RangeError: Operation cannot be trusted!
safeOp(
  9007199254740993 - 9007199254740995,
  9007199254740993,
  9007199254740995
)
// <- RangeError: Operation cannot be trusted!
safeOp(1 + 2, 1, 2)
// <- 3
```

这就是 `Number` 的所有情况，但我们还没有完成与数学相关的改进。接下来我们将注意力转向 `Math` 内置。

7.2 Math

ES6 为 `Math` 内置引入了很多新的静态方法。其中的一些方法是专门为使 C 语言更容易编译成 JavaScript 而设计的，你很少需要将它们用于日常的 JavaScript 应用开发。其他方法是对现有的舍入、取幂和三角函数 API 接口的补充。

我们开始吧。

7.2.1 Math.sign

许多语言中都有一个数学的 `sign` 方法，它返回一个矢量（`-1`、`0` 或 `1`）来表示所提供的输入的符号。JavaScript 中的 `Math.sign` 方法也是如此。但是，JavaScript 中的此方法还有两个可能的返回值：`-0` 和 `NaN`。查看以下代码段中的示例。

```
Math.sign(1) // <- 1
Math.sign(0) // <- 0
Math.sign(-0) // <- -0
Math.sign(-30) // <- -1
Math.sign(NaN) // <- NaN
Math.sign('one') // <- NaN，因为Number('one')值为NaN
Math.sign('0') // <- 0，因为Number('0')值为0
Math.sign('7') // <- 1，因为Number('7')值为7
```

注意 `Math.sign` 是如何将其输入转换为数字值的。虽然引入 `Number` 内置的方法不会通过 `Number(value)` 进行类型转换，但添加到 `Math` 中的大多数方法是具有这个特性的，正如我们将看到的那样。

7.2.2 Math.trunc

JavaScript 中已经有 `Math.floor` 和 `Math.ceil` 方法，我们可以用它们分别向下或向上取整。现在我们还有 `Math.trunc` 作为替代方案，该方法会舍弃小数部分但不进行舍入。这里输入也会被 `Number(value)` 强制转换为数值。

```
Math.trunc(12.34567) // <- 12
Math.trunc(-13.58) // <- -13
Math.trunc(-0.1234) // <- -0
Math.trunc(NaN) // <- NaN
Math.trunc('one') // <- NaN，因为Number('one')值为NaN
Math.trunc('123.456') // <- 123，Number('123.456')值为123.456
```

为 `Math.trunc` 创建一个简单的 ponyfill 来检查输入值是否大于零，并应用 `Math.floor` 或 `Math.ceil` 方法，如下所示。

```
function mathTrunc(value) {
  return value > 0 ? Math.floor(value) : Math.ceil(value)
}
```

7.2.3 Math.cbrt

`Math.cbrt` 方法是“立方根”的简称，类似于 `Math.sqrt` 是“平方根”的缩写，使用方法如下所示。

```
Math.cbrt(-1) // <- -1
Math.cbrt(3) // <- 1.4422495703074083
Math.cbrt(8) // <- 2
Math.cbrt(27) // <- 3
```

注意，此方法也会将非数值强制转换为数字。

```
Math.cbrt('8') // <- 2，因为Number('8')值为8
Math.cbrt('one') // <- NaN，因为Number('one')值为NaN
```

我们继续。

7.2.4 Math.expm1

这个操作用于计算 e 的 value 次幂的结果减 1。在 JavaScript 中，常量 e 被定义为 Math.E。以下代码段中的函数大致相当于 Math.expm1。

```
function expm1(value) {
  return Math.pow(Math.E, value) - 1
}
```

e^value^ 操作也可以表示为 Math.exp(value)。

```
function expm1(value) {
  return Math.exp(value) - 1
}
```

注意，Math.expm1 比单纯的 Math.exp(value) - 1 计算具有更高的精确度，应该作为首选来使用。

```
expm1(1e-20)
// <- 0
Math.expm1(1e-20)
// <- 1e-20
expm1(1e-10)
// <- 1.000000082740371e-10
Math.expm1(1e-10)
// <- 1.00000000005e-10
```

Math.expm1 的逆函数是 Math.log1p。

7.2.5 Math.log1p

这是 value+1 的自然对数——ln(value + 1)——也是 Math.expm1 的逆函数。在 JavaScript 中，一个数字的自然对数可以用 Math.log 表示。

```
function log1p(value) {
  return Math.log(value + 1)
}
```

与 Math.expm1 一样，Math.log1p 方法比手动执行 Math.log(value + 1) 操作更精确。

```
log1p(1.00000000005e-10)
// <- 1.000000082690371e-10
```

```
Math.log1p(1.00000000005e-10)
// <- 1e-10，就是Math.expm1(1e-10)的逆运算
```

7.2.6 Math.log10

以 10 为底的 `log10(value)` 的对数。

```
Math.log10(1000)
// <- 3
```

你可以使用 `Math.LN10` 常量为 `Math.log10` 创建 ponyfill。

```
function mathLog10(value) {
  return Math.log(x) / Math.LN10
}
```

接下来就是 `Math.log2`。

7.2.7 Math.log2

以 2 为底的 `log2(value)` 的对数。

```
Math.log2(1024)
// <- 10
```

你可以使用 `Math.LN2` 常量为 `Math.log2` 创建 ponyfill。

```
function mathLog2(value) {
  return Math.log(x) / Math.LN2
}
```

注意，ponyfill 版本不会像 `Math.log2` 那样精确，如下所示。

```
Math.log2(1 << 29) // 原生的实现方式
// <- 29
mathLog2(1 << 29) // ponyfill的实现方式
// <- 29.000000000000004
```

<< 运算符执行“按位左移”。在此操作中，按照操作的右边显示的，二进制表示的字节会从其左边的数字向左移动指定的位数。以下两个示例展示了如何使用 7.1.1 节介绍的二进制文字符号进行移位。

```
0b00000001 // 1
0b00000001 << 2 // 左移2位
0b00000100 // 4

0b00001101 // 1
0b00001101 << 4 // 左移4位
0b11010000 // 208
```

7.2.8　三角函数

ES6 中的 Math 对象添加了三角函数：

- Math.sinh(value) 返回 value 的双曲正弦值；
- Math.cosh(value) 返回 value 的双曲余弦值；
- Math.tanh(value) 返回 value 的双曲正切值 ；
- Math.asinh(value) 返回 value 的反曲正弦值 ；
- Math.acosh(value) 返回 value 的反曲余弦值 ；
- Math.atanh(value) 返回 value 的反曲正切值 。

7.2.9　Math.hypot

我们用 Math.hypot 返回每个参数的平方和的平方根。

```
Math.hypot(1, 2, 3)
// <- 3.741657386773941, (1*1 + 2*2 + 3*3)的平方根
```

通过手动执行这些操作，我们可以为 Math.hypot 创建 ponyfill。我们可以用 Math.sqrt 来计算平方根，并结合 Array#reduce 与扩展运算对平方进行求和[3]。

```
function mathHypot(...values) {
  const accumulateSquares (total, value) =>
    total + value * value
  const squares = values.reduce(accumulateSquares, 0)
  return Math.sqrt(squares)
}
```

令人惊讶的是，在这个特定用例下，手动创建的函数竟然比原生函数更精确。在以下代码示例中，我们可以看到手动创建的 hypot 函数提供了多一位的小数精确度。

```
Math.hypot(1, 2, 3) // 原生的实现方式
// <- 3.741657386773941
mathHypot(1, 2, 3) // ponyfill的实现方式
// <- 3.7416573867739413
```

7.2.10　按位计算助手

7.2 节的开头说过，一些 Math 的新方法是专门为将 C 语言更轻松地编译为 JavaScript 而设计的。它们是我们将要介绍的最后 3 种方法，用于处理 32 位数字。

1. Math.clz32

此方法的名称是“计算二进制表示的 32 位数字中的前导零位”的首字母缩写词。记住，

注 3：你可以阅读作者的 *Fun with Native Arrays* 一文来深入了解 Array 的方法。

`<<` 运算符执行“按位左位移”，我们来看看以下代码段中的 `Math.clz32` 的输入和输出。

```
Math.clz32(0) // <- 32
Math.clz32(1) // <- 31
Math.clz32(1 << 1) // <- 30
Math.clz32(1 << 2) // <- 29
Math.clz32(1 << 29) // <- 2
Math.clz32(1 << 31) // <- 0
```

2. `Math.imul`

返回类 C 语言中的 32 位乘法的结果。

3. `Math.fround`

将 `value` 舍入到最接近的 32 位浮点型表示的数字。

7.3 字符串和Unicode

你可能会想起 2.5 节中介绍过的模板字面量，以及如何使用它们来混合字符串和变量，或使用任何有效的 JavaScript 表达式，以产生字符串输出。

```
function greet(name) {
  return `Hello, ${ name }!`
}
greet('Gandalf')
// <- 'Hello, Gandalf!'
```

在 ES6 中，除了模板字面量语法外，字符串还有许多新的方法。这些方法可以分类为字符串操作方法和与 Unicode 相关的方法。我们从前者开始。

7.3.1 `String#startsWith`

在 ES6 问世前，无论何时想要检查一个字符串是否以其他字符串开头，我们都会使用 `String#indexOf` 方法，如下所示。结果 `0` 意味着字符串以提供的值开始。

```
'hello gary'.indexOf('gary')
// <- 6
'hello gary'.indexOf('hello')
// <- 0
'hello gary'.indexOf('stephan')
// <- -1
```

要想检查一个字符串是否以另一个字符串开始，你可以用 `String#indexOf` 比较它们并检查找到的索引值是否为字符串开头的 `0` 索引值。

```
'hello gary'.indexOf('gary') === 0
// <- false
'hello gary'.indexOf('hello') === 0
```

```
// <- true
'hello gary'.indexOf('stephan') === 0
// <- false
```

现在你可以使用 String#startsWith 方法，以避免因检查索引是否为 0 而带来不必要的复杂性。

```
'hello gary'.startsWith('gary')
// <- false
'hello gary'.startsWith('hello')
// <- true
'hello gary'.startsWith('stephan')
// <- false
```

为了确定一个字符串是否包含一个从特定索引值开始的字符串，我们可以用 String#indexOf 先获取该字符串的一部分。

```
'hello gary'.slice(6).indexOf('gary') === 0
// <- true
```

我们不能简单地检查索引是否为 6，如果在达到索引值 6 前找到查询值，那么会产生假阴性。以下示例表明，即使要查询的 'ell' 字符串确实处于索引值 6，但是只比较 String#indexOf 的结果和 6 并不足以获得正确的结果。

```
'hello ell'.indexOf('ell') === 6
// <- false，因为结果是1
```

我们可以使用 indexOf 中的 startIndex 参数来解决这个问题，无须依赖于 String#slice。注意，这里我们仍然与 6 进行比较，因为字符串在开始的操作中未被切分。

```
'hello ell'.indexOf('ell', 6) === 6
// <- true
```

对于正在搜索的内容，与其关注所有这些字符串实现搜索的细节并编写如何搜索的代码，不妨使用 String#startsWith 来传递可选 startIndex 参数。

```
'hello ell'.startsWith('ell', 6)
// <- true
```

7.3.2 String#endsWith

这个方法与 String#startsWith 是镜像的，正如 String#lastIndexOf 是 String#indexOf 的镜像。它告诉我们一个字符串是否以另一个字符串结尾。

```
'hello gary'.endsWith('gary')
// <- true
'hello gary'.endsWith('hello')
// <- false
```

与 `String#startsWith` 相反，此方法有一个索引值，用于指示查找应该结束的位置，而不是它应该开始的位置。它默认为字符串的长度。

```
'hello gary'.endsWith('gary', 10)
// <- true
'hello gary'.endsWith('gary', 9)
// <- false，本例实际上以gar结尾（因为第9位是r，而非y）
'hello gary'.endsWith('hell', 4)
// <- true
```

`String#includes` 是可以简化 `String#indexOf` 特定用例的最后一种方法。

7.3.3 String#includes

我们可以用 `String#includes` 来确定一个字符串是否包含另一个字符串，如所示。

```
'hello gary'.includes('hell')
// <- true
'hello gary'.includes('ga')
// <- true
'hello gary'.includes('rye')
// <- false
```

这与 ES5 用例中的 `String#indexOf` 测试的结果 `-1` 是等价的，用于检查要搜索的字符串是否在任何位置被找到，如下所示。

```
'hello gary'.indexOf('ga') !== -1
// <- true
'hello gary'.indexOf('rye') !== -1
// <- false
```

你还可以为 `String#includes` 提供索引值，搜索将从该索引值开始。

```
'hello gary'.includes('ga', 4)
// <- true
'hello gary'.includes('ga', 7)
// <- false
```

我们来看看不止是 `String#indexOf` 替代方案的一些方法。

7.3.4 String#repeat

这种方便的方法允许你将一个字符串重复 `count` 次。

```
'ha'.repeat(1)
// <- 'ha'
'ha'.repeat(2)
// <- 'haha'
'ha'.repeat(5)
```

```
// <- 'hahahahaha'
'ha'.repeat(0)
// <- ''
```

提供的 count 应该是一个非负的有限数字。

```
'ha'.repeat(Infinity)
// <- RangeError
'ha'.repeat(-1)
// <- RangeError
```

小数值会向下取整到最近的整数。

```
'ha'.repeat(3.9)
// <- 'hahaha'，向下取整为3
```

使用 NaN 时，count 会被解析为 0。

```
'ha'.repeat(NaN)
// <- ''
```

非数字值会被强制转换为数字。

```
'ha'.repeat('ha')
// <- '，因为Number('ha')值为NaN
'ha'.repeat('3')
// <- 'hahaha'，因为Number('3')值为3
```

(-1, 0) 范围内的值会四舍五入到 -0，因为 count 是通过 ToInteger 方法计算的，正如规范中记录的那样[4]。规范中的这一步要求 count 使用以下代码中的公式进行转换。

```
function ToInteger(number) {
  return Math.floor(Math.abs(number)) * Math.sign(number)
}
```

ToInteger 函数会将 (-1, 0) 范围内的任何值转换为 -0。因此，向 String#repeat 方法传递 (-1, 0) 范围中的数字将被视为零，而 [-1, -Infinity) 范围中的数字将导致异常，正如我们前面所了解的那样。

```
'na'.repeat(-0.1)
// <- '，因为向下取整为-0
'na'.repeat(-0.9)
// <- '，因为向下取整为-0
'na'.repeat(-0.9999)
// <- '，因为向下取整为-0
'na'.repeat(-1)
// <- Uncaught RangeError: Invalid count value
```

注 4：在 ECMAScript 6 规范中，String#repeat 位于 21.1.3.13 节。

String#repeat 的典型用例可能是填充函数。以下代码段中的 indent 函数使用多行字符串，并用默认的两个空格缩进每行所需的 spaces。

```
function indent(text, spaces = 2) {
  return text
    .split('\n')
    .map(line => ' '.repeat(spaces) + line)
    .join('\n')
}

indent(`a
b
c`, 2)
// <- '  a\n  b\n  c'
```

7.3.5 字符串填充和去空白

撰写本书时，ES2017 发布了两种新的字符串填充方法：String#padStart 和 String#padEnd。使用这些方法后，我们就无须实现类似前面代码段中的 indent 的内容了。在执行字符串操作时，我们经常要填充一个字符串，以便与我们想要的格式保持一致。在格式化数字、货币、HTML，以及经常涉及的等宽文本的各种情况中，这很有用。

使用 padStart 时，我们将为目标字符串和填充字符串指定所需要的长度，默认为单个空格字符。如果原始字符串至少与指定长度相同，那么 padStart 会导致空操作，并返回原始字符串。

在以下示例中，被填充的字符串的期望长度为 5，而原始字符串的长度已经为 5，因此返回值不变。

```
'01.23'.padStart(5)
// <- '01.23'
```

以下示例中的原始字符串的长度为 4，因此 padStart 在字符串的开头添加了一个空格，以便长度达到期望值 5。

```
'1.23'.padStart(5)
// <- ' 1.23'
```

以下示例与前一个示例类似，只是它用 '0' 填充，而不是缺省值 ' '。

```
'1.23'.padStart(5, '0')
// <- '01.23'
```

注意，padStart 将持续填充字符串，直到达到最大长度。

```
'1.23'.padStart(7, '0')
// <- '0001.23'
```

但是，如果填充字符串过长，那么可能会被截断。提供的长度是填充字符串的最大长度，除非原始字符串已经比该长度长。

```
'1.23'.padStart(7, 'abcdef')
// <- 'abc1.23'
```

padEnd 方法具有类似的 API，但它是在原始字符串的末尾添加填充，而不是开头。以下代码段说明了这种差异。

```
'01.23'.padEnd(5) // <- '01.23'
'1.23'.padEnd(5) // <- '1.23 '
'1.23'.padEnd(5, '0') // <- '1.230'
'1.23'.padEnd(7, '0') // <- '1.23000'
'1.23'.padEnd(7, 'abcdef') // <- '1.23abc'
```

撰写本书时，已经有一个处于阶段 2 的有关字符串修整的提案，其中包含 String#trimStart 和 String#trimEnd 方法。我们可以用 trimStart 删除字符串开头的任何空格，并用 trimEnd 删除字符串末尾的任何空格。

```
'   this should be left-aligned   '.trimStart()
// <- 'this should be left-aligned   '
'   this should be right-aligned   '.trimEnd()
// <- '   this should be right-aligned'
```

接下来我们将切换协议并学习 Unicode。

7.3.6 Unicode

JavaScript 字符串可以用 UTF-16 代码单元表示[5]。每个代码单元可用于表示 [U+0000, U+FFFF] 范围中的代码点，也称为"基本多文种平面"（BMP，basic multilingual plane）。你可以用 '\u3456' 语法在 BMP 平面中表示各个代码点 , 也可以用 \x00..\xff 符号来表示 [U+0000, U+0255] 范围内的代码单元。举例来说，'\xbb' 代表 '»' 的 U+00BB 代码点，你也可以通过 String.fromCharCode(0xbb) 进行验证。

对于超出 U+FFFF 的代码点，你可以用代理对表示它们，也就是两个连续的代码单元。例如，马表情（🐎）的代码点可以用 '\ud83d\udc0e' 的连续代码单元来表示。在 ES6 中，你也可以用 '\u{1f40e}' 符号表示代码点（该示例也是马表情符号）。

注意，其内部表示没有改变，因此该单个代码点之后仍然有两个代码单元。实际上，'\u{1f40e}'.length 的值为 2，每个代码单元是一个长度。

以下代码中的 '\ud83d\udc0e\ud83d\udc71\u2764' 字符串是由好几个表情符号构成的。

注 5：要想了解更多相关信息，可以参见 *UCS-2, UCS-4, UTF-16 and UTF-32*（https://unicodebook.readthedocs.io/unicode_encodings.html#ucs-2-ucs-4-utf-16-and-utf-32）。

```
'\ud83d\udc0e\ud83d\udc71\u2764'
// <- '🐎👱❤'
```

虽然该字符串由 5 个代码单元组成，但我们知道长度应该为 3，因为只有 3 个表情符号。

```
'\ud83d\udc0e\ud83d\udc71\u2764'.length
// <-  5
'🐎👱❤'.length
```

在 ES6 问世前，计算代码点是非常棘手的，因为该语言没有在 Unicode 方面做出任何努力，如以下代码段中的 `Object.keys` 所示。它会为我们的 3 个表情符号字符串返回 5 个键，因为这 3 个代码点总共使用了 5 个代码单元。

```
Object.keys('🐎👱❤')
// <-  ['0', '1', '2', '3', '4']
```

如果现在思考一下 `for` 循环，那么我们可以更清楚地观察到这是一个问题。以下示例想从 `text` 字符串中提取每个单独的表情符号，但我们得到了每个代码单元，而不是它们形成的代码点。

```
const text ='🐎👱❤'
for (let i = 0; i < text.length; i++) {
  console.log (text[i])
  // <- '\ud83d'
  // <- '\udc0e'
  // <- '\ud83d'
  // <- '\udc71'
  // <- '\u2764'
}
```

好在 ES6 中的字符串遵循迭代协议。我们可以用字符串迭代器来遍历代码点，即使这些代码点是由代理对构成的。

7.3.7 `String.prototype[Symbol.iterator]`

考虑到代码单元循环的问题，字符串迭代器迭代产生的是代码点。

```
for (const codePoint of'🐎👱❤') {
  console.log(codePoint)
  // <- '🐎'
  // <- '👱'
  // <- '❤'
}
```

正如我们前面看到的那样，用 `String#length` 根据代码点测量字符串的长度是不行的，因为它计算的是代码单位。但是，我们可以用迭代器将字符串拆分为其代码点，就像我们在 `for..of` 示例中所做的那样。

我们可以使用依赖于迭代器协议的扩展运算符将字符串拆分为由其符合的代码点组成的数组，然后取出该数组的 `length`，从而获得正确的代码点数，如下所示。

```
[...'🐎👨🏾❤'] .length
// <-  3
```

记住，如果想要对字符串长度进行 100%的精确计算，那么将字符串拆分为代码点是不够的。如以 `\u0305` 代表的上划线和 Unicode 代码单元的组合，这个代码单元本身就是一个上划线，如下所示。

```
'\u0305'
// <- '̅'
```

然而，当以另一个代码单元为前缀时，它们被组合成了单个字形。

```
function overlined(text) {
  return '${ text }\u0305'
}

overlined('o')
// <- 'o̅'
'hello world'.split('').map(overlined).join('')
// <- 'h̅e̅l̅l̅o̅ ̅w̅o̅r̅l̅d̅'
```

事实证明，试图通过计算代码点来计算出实际长度是不够的，就像使用 `String#length` 计算代码点时那样，如下所示。

```
'o̅'.length
// <- 2
[...'o̅'].length
// <- 2，应为1
[...'h̅e̅l̅l̅o̅ ̅w̅o̅r̅l̅d̅'].length
// <- 22，应为11
[...'h̅e̅l̅l̅o̅ world'].length
// <- 16，应为11
```

正如 Unicode 专家 Mathias Bynens 指出的那样，通过代码点进行分割是不够的。与前面示例中使用的表情等代理对不同，字符串迭代器不考虑其他字形群集[6]。运气不好的情况下，我们还必须使用正则表达式或实用工具库来计算正确的字符串长度。

7.3.8　有关分割字形段的提案

将多个代码点合并为一个可视化字形变得越来越普遍[7]。一个新的提案正在进行中（目前处

注 6：建议你阅读一下 Mathias Bynens 的 *JavaScript Has A Unicode Problem* 一文。Mathias 在文中分析了 JavaScript 与 Unicode 的关系。

注 7：表情符号使字形变得普遍，它有时使用由 4 个代码点组成的字形。多个代码点组成的表情符号列表参见 http://unicode.org/emoji/charts-beta/emoji-zwj-sequences.html。

于阶段 2)，该提案可以一次解决迭代字形集群的问题。它引入了一个 `Intl.Segmenter` 内置，用于将字符串拆分为可迭代的序列。

要想使用 `Segmenter` API，首先需要创建一个 `Intl.Segmenter` 实例，以指定我们想要的语言环境和粒度级别：每个字形、单词、句子或行。分段器实例可以为任何给定的字符串生成迭代器，并按指定的 `granularity` 方式对其进行分割。注意，分段算法可能因语言环境而异，这就是为什么它是 API 的一部分。

以下示例定义了一个 `getGraphemes` 函数，该函数会为任何给定的语言环境和文本片段生成一组字形集群。

```
function getGraphemes(locale, text) {
  const segmenter = new Intl.Segmenter(locale, {
    granularity: 'grapheme'
})
  const sequence = segmenter.segment(text)
  const graphemes = [...sequence].map(item => item.segment)
  return graphemes
}
getGraphemes('es', 'Esto está bien bueno!')
```

使用 `Segmenter`（https://github.com/tc39/proposal-intl-segmenter）提案后，我们就不会在分割包含表情符号或其他组合代码单元的字符串时遇到任何问题了。

我们来看看 ES6 中引入的与 Unicode 相关的其他方法。

7.3.9 String#codePointAt

我们可以用 `String#codePointAt` 来获取字符串中给定位置的代码点的数字表示。注意，预期的起始位置是代码单元的索引，而不是代码点。在以下示例中，我们为字符串中的每个表情符号（🐎👴❤）打印了代码点。

```
const text = '\ud83d\udc0e\ud83d\udc71\u2764'
text.codePointAt(0)
// <- 0x1f40e
text.codePointAt(2)
// <- 0x1f471
text.codePointAt(4)
// <- 0x2764
```

识别需要提供给 `String#codePointAt` 的索引可能会很麻烦，这就是为什么你应该循环遍历一个字符串迭代器来代替你自己识别它们。然后你可以为序列中的每个代码点调用 `.codePointAt(0)`，并且 `0` 始终是正确的起始索引。

```
const text = '\ud83d\udc0e\ud83d\udc71\u2764'
for (const codePoint of text) {
  console.log(codePoint.codePointAt(0))
```

```
  // <- 0x1f40e
  // <- 0x1f471
  // <- 0x2764
}
```

我们还可以用扩展运算符和 `Array#map` 的组合将示例简化为一行代码。

```
const text = '\ud83d\udc0e\ud83d\udc71\u2764'
[...text].map(cp => cp.codePointAt(0))
// <- [0x1f40e, 0x1f471, 0x2764]
```

你可以采用 base-16 表示形式来替代 base-10 代码点，并用新的 Unicode 代码点来创建一个字符串，从而避免使用 `\u{codePoint}` 语法。此语法允许你表示超出 BMP 的 Unicode 代码点。也就是说，`[U+0000, U+FFFF]` 范围之外的代码点通常用 `\u1234` 语法表示。

我们从更新示例开始，以打印十六进制版本的代码点。

```
const text = '\ud83d\udc0e\ud83d\udc71\u2764'
[...text].map(cp => cp.codePointAt(0).toString(16))
// <- ['1f40e', '1f471', '2764']
```

我们可以将这些 base-16 的值包含在 `'\u{codePoint}'` 中，并再次获得表情符号值。

```
'\u{1f40e}'
// <- '🐎'
'\u{1f471}'
// <- '👱'
'\u{2764}'
// <- '❤'
```

7.3.10 String.fromCodePoint

该方法接收一个数字并返回一个代码点。注意我是如何在刚通过 `String#codePointAt` 获得的简洁 base-16 代码点中使用前缀 `0x` 的。

```
String.fromCodePoint(0x1f40e)
// <- '🐎'
String.fromCodePoint(0x1f471)
// <- '👱'
String.fromCodePoint(0x2764)
// <- '❤'
```

你也可以使用普通的 base-10 字面量，并获得相同的结果。

```
String.fromCodePoint(128014)
// <- '🐎'
String.fromCodePoint(128113)
// <- '👱'
String.fromCodePoint(10084)
// <- '❤'
```

你可以根据需要向 `String.fromCodePoint` 传入尽可能多的代码点。

```
String.fromCodePoint(0x1f40e, 0x1f471, 0x2764)
// <- '🐎👱❤'
```

作为徒劳无益的练习，我们可以将一个字符串映射到代码点的数字表示，然后返回到代码点本身。

```
const text ='\ud83d\udc0e\ud83d\udc71\u2764'
[...text]
  .map(cp => cp.codePointAt(0))
  .map(cp => String.fromCodePoint(cp))
  .join('')
// <- '🐎👱❤'
```

反转字符串也可能会导致问题。

7.3.11　Unicode-Aware字符串反转

思考以下这段代码。

```
const text = '\ud83d\udc0e\ud83d\udc71\u2764'
text.split('').map(cp => cp.codePointAt(0))
// <- [55357, 56334, 55357, 56433, 10084]
text.split('').reverse().map(cp => cp.codePointAt(0))
// <- [10084, 56433, 128014, 55357]
```

问题是，当我们不得不为了正确的解决方案而反转代码点时，我们反转的是各个代码单元。相反，如果用扩展运算符将字符串按其代码点拆分，然后将其颠倒，那么代码点将被保留，并且字符串将被正确地颠倒过来。

```
const text ='\ud83d\udc0e\ud83d\udc71\u2764'
[...text].reverse().join('')
// <- '❤👱🐎'
```

这样一来，我们就不会破坏代码点了。再次强调，这不适用于所有字形集群。

```
[...'hello\u0305'].reverse().join('')
// <- `̄olleh`
```

我们将要介绍的与 Unicode 相关的最后一个方法是 `.normalize`。

7.3.12　String#normalize

有些字符串的代码点不同，可看上去却是完全一样的。思考以下示例，其中看起来相同的两个字符串在任何 JavaScript 运行时都不会被视为相同。

```
'mañana' === 'mañana'
// <- false
```

这里发生了什么？左边的版本有一个 ñ，右边的版本有一个合成的字符 +˜` 和一个 n。两者在视觉上相同，但如果查看代码点，我们就会发现它们是不同的。

```
[...'mañana'].map(cp => cp.codePointAt(0).toString(16))
// <- ['6d', '61', 'f1', '61', '6e', '61']
[...'mañana'].map(cp => cp.codePointAt(0).toString(16))
// <- ['6d', '61', '6e', '303', '61', '6e', '61']
```

就像 'hellō' 示例那样，虽然两个字符串的长度看上去都是 6，但实际上第二个字符串的长度是 7。

```
[...'mañana'].length
// <- 6
[...'mañana'].length
// <- 7
```

如果用 String#normalize 对第二个版本进行标准化，那么将返回与第一个版本中相同的代码点。

```
const normalized = 'mañana'.normalize()
[...normalized].map(cp => cp.codePointAt(0).toString(16))
// <- ['6d', '61', 'f1', '61', '6e', '61']
normalized.length
// <- 6
```

注意，要想测试相等性，我们应该在比较它们时对两个字符串都使用 String#normalize。

```
function compare(left, right) {
  return left.normalize() === right.normalize()
}
const normal = 'mañana'
const irregular = 'mañana'
normal === irregular
// <- false
compare(normal, irregular)
// <- true
```

7.4　正则表达式

本节将探讨 ES6 及后续版本中的正则表达式。ES6 引入了一些正则表达式修饰符：/y（粘连修饰符）及 /u（Unicode 修饰符）。接下来我们将讨论 TC39 上通过 ECMAScript 规范开发流程的 5 个提案。

7.4.1　粘连修饰符/y

ES6 中引入了粘连修饰符 y，该修饰符与全局修饰符 g 相似。同全局正则表达式一样，粘连修饰符通常用于多次匹配，直到输入字符串耗尽为止，且粘连正则表达式会将

lastIndex 移到最后一次匹配的位置。唯一的区别是，粘连正则表达式必须确保匹配从剩余的第一个位置开始，而全局正则表达式只要确保剩余位置存在匹配即可。

以下示例清楚地展示了两者间的区别。当输入字符串为 'haha haha haha' 且正则表达式为 /ha/ 时，全局修饰符将在每次出现 'ha' 时匹配，而粘连修饰符将会匹配前两个，之所以这样，是因为第三次出现 'ha' 的起始索引是 5，而不是 4。

```
function matcher(regex, input) {
  return () => {
    const match = regex.exec(input)
    const lastIndex = regex.lastIndex
    return { lastIndex, match }
  }
}
const input = 'haha haha haha'
const nextGlobal = matcher(/ha/g, input)
console.log(nextGlobal()) // <- { lastIndex: 2, match: ['ha'] }
console.log(nextGlobal()) // <- { lastIndex: 4, match: ['ha'] }
console.log(nextGlobal()) // <- { lastIndex: 7, match: ['ha'] }
const nextSticky = matcher(/ha/y, input)
console.log(nextSticky()) // <- { lastIndex: 2, match: ['ha'] }
console.log(nextSticky()) // <- { lastIndex: 4, match: ['ha'] }
console.log(nextSticky()) // <- { lastIndex: 0, match: null }
```

如果强行移动 lastIndex，就可以验证粘连匹配器是可以工作的，如下所示。

```
const rsticky = /ha/y
const nextSticky = matcher(rsticky, input)
console.log(nextSticky()) // <- { lastIndex: 2, match: ['ha'] }
console.log(nextSticky()) // <- { lastIndex: 4, match: ['ha'] }
rsticky.lastIndex = 5
console.log(nextSticky()) // <- { lastIndex: 7, match: ['ha'] }
```

作为提高编译器中的词法分析器性能的一种方式添加到 JavaScript 中，粘连匹配很大程度上要依赖于正则表达式。

7.4.2 Unicode修饰符/u

ES6 还引入了一个 u 修饰符。u 代表 Unicode 模式，这个修饰符可以视为更严格版的正则表达式。

以下正则表达式中包含了一个无须转义的字符 'a'，不使用 u 修饰符时不会报错。

```
/\a/.test('ab')
// <- true
```

但对字符 'a'（非正则保留字符）进行转义处理，同时使用 u 修饰符，就会报错，如下所示。

```
/\a/u.test('ab')
// <- SyntaxError: Invalid escape: /\a/
```

通过 ES6 中新引入的 Unicode 字符表示法，以下示例试图在正则表达式中嵌入马表情符号 `'\u{1f40e}'`，但正则表达式无法匹配马表情符号。如果不使用 `u` 修饰符，`\u{...}` 这种形式会被认为是包含无须转义字符 `u` 的序列。

```
/\u{1f40e}/.test('🐎')  // <- false
/\u{1f40e}/.test('u{1f40e}') // <- true
```

`u` 修饰符引入了对 Unicode 代码点转义的支持，从而支持在正则表达式中使用表情符号，如 `\u{1f40e}` 这样的马表情符号。

```
/\u{1f40e}/u.test（'🐎'）
// <- true
```

在不使用 `u` 修饰符时，`.` 模式可以匹配除终止符以外的任何 BMP 符号。以下示例测试 `U+1D11E MUSICAL SYMBOL G CLEF`，这是一个在普通正则表达式中无法匹配 `.` 模式的星状符号。

```
const rdot = /^.$/
rdot.test('a') // <- true
rdot.test('\n') // <- false
rdot.test('\u{1d11e}') // <- false
```

使用 `u` 修饰符时，不在 BMP 中的 Unicode 符号也可以被匹配。以下示例表明，使用 `u` 修饰符后星状符号也可以匹配 `.` 模式。

```
const rdot = /^.$/u
rdot.test('a') // <- true
rdot.test('\n') // <- false
rdot.test('\u{1d11e}') // <- true
```

使用 `u` 修饰符时，这种 Unicode 的严格模式也可以应用于量词和字符类，其中 Unicode 代码点将被视为单个符号，而不是仅匹配其中第一个代码单元。`u` 修饰符和不区分大小写的 `i` 修饰符同用时会遵循 Unicode 的大写转换规则，以规范输入字符串和正则表达式中的代码点 [8]。

7.4.3　具名捕获组

如今，JavaScript 正则表达式可以在编号捕获组和非捕获组中对匹配进行分组。在以下代码中，我们使用几个组从包含由 `'='` 分隔的键 / 值对的输入字符串中提取键和值。

```
function parseKeyValuePair(input) {
  const rattribute = /([a-z]+)=([a-z]+)/
```

注 8：要想了解更多关于正则表达式中 `u` 修饰符的细节，可以阅读一下 Mathias Bynens 的文章 *Unicode-aware Regular Expressions in ES2015*。

```
  const [, key, value] = rattribute.exec(input)
  return { key, value }
}
parseKeyValuePair('strong=true')
// <- { key: 'strong', value: 'true' }
```

虽然非捕获组会被丢弃且不显示在最终结果中，但对匹配仍然有用。除了支持由 `'='` 分隔的键 / 值对输入外，以下示例还支持由 `'is'` 分隔的键 / 值对输入。

```
function parseKeyValuePair(input) {
  const rattribute = /([a-z]+)(?:=|\sis\s)([a-z]+)/
  const [, key, value] = rattribute.exec(input)
  return { key, value }
}
parseKeyValuePair('strong is true')
// <- { key: 'strong', value: 'true' }
parseKeyValuePair('flexible=too')
// <- { key: 'flexible', value: 'too' }
```

虽然前面示例中的数组解构隐藏了代码对数组索引的依赖，但事实是，无论如何匹配都会被放置在有序数组中。具名捕获组提案[9]（撰写本书时处于阶段 3）增加了类似于 `(?<groupName>)` 的语法到 Unicode 正则表达式，通过这种方式为捕获组命名，就可以在返回的匹配对象中获得 `groups` 属性了。调用 `RegExp#exec` 或 `String#match` 时，可以从结果对象中解构 `groups` 属性。

```
function parseKeyValuePair(input) {
  const rattribute = (
    /(?<key>[a-z]+)(?:=|\sis\s)(?<value>[a-z]+)/
)
  const { groups } = rattribute.exec(input)
  return groups
}
parseKeyValuePair('strong=true')
// <- { key: 'strong', value: 'true' }
parseKeyValuePair('flexible=too')
// <- { key: 'flexible', value: 'too' }
```

JavaScript 正则表达式支持反向引用，其中捕获组可以复用于查找重复项。以下代码段对第一个捕获组使用反向引用来识别 `'user:password'` 输入中的用户名和密码相同的情况。

```
function hasSameUserAndPassword(input) {
  const rduplicate = /([^:]+):\1/
  return rduplicate.exec(input) !== null
}
hasSameUserAndPassword('root:root') // <- true
hasSameUserAndPassword('root:pF6GGlyPhoy1!9i') // <- false
```

具名捕获组提案增加了对具名反向引用的支持，以反向引用具名捕获组。

注 9：参见具名捕获组提案文档（https://github.com/tc39/proposal-regexp-named-groups）。

```
function hasSameUserAndPassword(input) {
  const rduplicate = /(?<user>[^:]+):\k<user>/u
  return rduplicate.exec(input) !== null
}
hasSameUserAndPassword('root:root') // <- true
hasSameUserAndPassword('root:pF6GGlyPhoy1!9i') // <- false
```

`\k<groupName>` 引用可以与编号引用串联使用，但在使用具名引用时最好避免使用编号引用。

最后，具名组可以在传递给 `String#replace` 的替换中直接被引用。在以下代码中，我们使用 `String#replace` 以及具名组将美国日期格式替换成匈牙利日期格式。

```
function americanDateToHungarianFormat(input) {
  const ramerican = (
    /(?<month>\d{2})\/(?<day>\d{2})\/(?<year>\d{4})/
)
  const hungarian = input.replace(
    ramerican,
    '$<year>-$<month>-$<day>'
)
  return hungarian
}
americanDateToHungarianFormat('06/09/1988')
// <- '1988-09-06'
```

如果 `String#replace` 的第二个参数是一个函数，那么可以通过参数列表末尾的一个名为 `groups` 的新参数来访问具名组。现在，该功能的签名是 `(match, ...captures, groups)`。在以下示例中，注意我们是如何使用与上个示例中找到的替换字符串类似的模板字面量。替换字符串遵循 `$<groupName>` 语法而非 `${ groupName }` 的事实表明，如果使用模板字面量，那么我们可以在替换字符串中对组进行命名，而不必诉诸转义码。

```
function americanDateToHungarianFormat(input) {
  const ramerican = (
    /(?<month>\d{2})\/(?<day>\d{2})\/(?<year>\d{4})/
)
  const hungarian = input.replace(ramerican, (...rest) => {
    const groups = rest[rest.length - 1]
    const { month, day, year } = groups
    return `${ year }-${ month }-${ day }`
  })
  return hungarian
}
americanDateToHungarianFormat('06/09/1988') // <- '1988-09-06'
```

7.4.4 Unicode属性转义

Unicode 属性转义提案[10]（目前处于阶段 3）描述的是一种新的转义序列，可用于标有 `u` 修饰符的正则表达式。该提案为二进制 Unicode 属性和非二进制 Unicode 属性分别添加了

注 10：参见 Unicode 属性转义提案文档（https://github.com/tc39/proposal-regexp-unicode-property-escapes）。

`\p{LoneUnicodePropertyNameOrValue}` 和 `\p{UnicodePropertyName=UnicodePropertyValue}` 的转义。`\P` 是 `\p` 的反向匹配，即匹配不满足条件的字符。

Unicode 标准为每个符号都定义了属性。有了这些属性，我们就可以对 Unicode 字符进行高级查询了。例如，希腊字母表中的符号的 `Script` 属性值为 `Greek`，利用这一点，我们可以使用新的转义符来匹配任何希腊文的 Unicode 符号。

```
function isGreekSymbol(input) {
  const rgreek = /^\p{Script=Greek}$/u
  return rgreek.test(input)
}
isGreekSymbol('π')
// <- true
```

或者我们也可以用 `\P` 来匹配非希腊的 Unicode 符号。

```
function isNonGreekSymbol(input) {
  const rgreek = /^\P{Script=Greek}$/u
  return rgreek.test(input)
}
isNonGreekSymbol('π')
// <- false
```

当需要对每个 Unicode 的十进制数符号进行匹配，而不只是像那样 `\d` 匹配 `[0-9]` 时，我们可以使用 `\p{Decimal_Number}`，如下所示。

```
function isDecimalNumber(input) {
  const rdigits = /^\p{Decimal_Number}+$/u
  return rdigits.test(input)
}
isDecimalNumber（'𝟏𝟐𝟑𝟜𝟝𝟞𝟩𝟪𝟫𝟬𝟭𝟮𝟯𝟺𝟻𝟼'）
// <- true
```

你可以查看支持的 Unicode 属性和值（https://github.com/mathiasbynens/regexpu-core/blob/master/property-escapes.md）中的详尽概述。

7.4.5　后行断言

长久以来，JavaScript 一直支持先行断言。该功能允许我们匹配一个表达式，前提是后面跟着另一个表达式。这些断言表达为 `(?=...)`。无论先行断言是否匹配，该匹配的结果都将被丢弃，并且不会消耗输入字符串的字符。

以下示例用先行断言来测试输入字符串是否有一串后面跟着 .js 的字母，在这种情况下，它会返回没有该 .js 部分的文件名。

```
function getJavaScriptFilename(input) {
  const rfile = /^(?<filename>[a-z]+)(?=\.js)\.[a-z]+$/u
```

```
  const match = rfile.exec(input)
  if (match === null) {
    return null
  }
  return match.groups.filename
}
getJavaScriptFilename('index.js') // <- 'index'
getJavaScriptFilename('index.php') // <- null
```

与先行肯定断言相反，先行否定断言的写法是 (?!...)，当表达式不匹配时，断言才会成功。以下代码使用了先行否定断言，我们可以观察到匹配结果恰恰相反：除了 .js 结尾的任何表达式都可以使断言通过。

```
function getNonJavaScriptFilename(input) {
  const rfile = /^(?<filename>[a-z]+)(?!\.js)\.[a-z]+$/u
  const match = rfile.exec(input)
  if (match === null) {
    return null
  }
  return match.groups.filename
}
getNonJavaScriptFilename('index.js') // <- null
getNonJavaScriptFilename('index.php') // <- 'index'
```

后行断言提案[11]（阶段 3）同时引入了肯定和否定的后行断言，分别用 (?<=...) 和 (?<!...) 表示。这些断言可以用于确认我们想要匹配的模式是否在另一个给定模式之后。以下代码使用后行肯定断言来匹配美元金额，而不是欧元金额。

```
function getDollarAmount(input) {
  const rdollars = /^(?<=\$)(?<amount>\d+(?:\.\d+)?)$/u
  const match = rdollars.exec(input)
  if (match === null) {
    return null
  }
  return match.groups.amount
}
getDollarAmount('$12.34') // <- '12.34'
getDollarAmount('€12.34') // <- null
```

反过来，后行否定断言可以用于匹配非美元金额的数字。

```
function getNonDollarAmount(input) {
  const rnumbers = /^(?<!\$)(?<amount>\d+(?:\.\d+)?)$/u
  const match = rnumbers.exec(input)
  if (match === null) {
    return null
  }
  return match.groups.amount
}
```

注 11：参见后行断言提案文档（https://github.com/tc39/proposal-regexp-lookbehind）。

```
getNonDollarAmount('$12.34') // <- null
getNonDollarAmount('€12.34') // <- '12.34'
```

7.4.6　新的/s（dotAll）修饰符

在使用 . 模式时，我们通常希望匹配每个独立字符。然而，JavaScript 中的 . 表达式不匹配星状字符（可以通过添加 u 修饰符来解决）或行终止符。

```
const rcharacter = /^.$/
rcharacter.test('a') // <- true
rcharacter.test('\t') // <- true
rcharacter.test('\n') // <- false
```

有时开发人员会使用变通的方式来合成与任何字符匹配的模式。以下代码中的表达式匹配任何字符，从而实现我们期望从 . 模式匹配器中获得的行为。

```
const rcharacter = /^[\s\S]$/
rcharacter.test('a') // <- true
rcharacter.test('\t') // <- true
rcharacter.test('\n') // <- true
```

dotAll 提案[12]（阶段 3）添加了 s 修饰符，该修饰符使得 JavaScript 正则表达式中的 . 可以匹配任何独立字符。

```
const rcharacter = /^.$/s
rcharacter.test('a') // <- true
rcharacter.test('\t') // <- true
rcharacter.test('\n') // <- true
```

7.4.7　String#matchAll

通常来说，当使用全局或粘连修饰符的正则表达式时，我们希望遍历每个匹配捕获组的集合。目前，生成匹配列表可能有点麻烦：我们需要在循环中用 String#match 或 RegExp#exec 来收集捕获组，直到正则表达式与 lastIndex 不匹配为止。以下示例中的 parseAttributes 生成器函数就是这么做的。

```
function* parseAttributes(input) {
  const rattributes = /(\w+)="([^"]+)"\s/ig
  while (true) {
    const match = rattributes.exec(input)
    if (match === null) {
      break
    }
    const [ , key, value] = match
    yield [key, value]
  }
```

注 12：参见 dotAll 修饰符提案文档（https://github.com/tc39/proposal-regexp-dotall-flag）。

```
}
const html = '<input type="email"
placeholder="hello@mjavascript.com" />'
console.log(...parseAttributes(html))
// [
//   ['type', 'email']
//   ['placeholder', 'hello@mjavascript.com']
// ]
```

这种方法的问题是，它是为正则表达式及其捕获组而定制的。现在，我们可以创建一个只关注遍历匹配并收集捕获组集合的 `matchAll` 生成器来解决该问题，如下所示。

```
function* matchAll(regex, input) {
  while (true) {
    const match = regex.exec(input)
    if (match === null) {
      break
    }
    const [ , ...captures] = match
    yield captures
  }
}
function* parseAttributes(input) {
  const rattributes = /(\w+)="([^"]+)"\s/ig
  yield* matchAll(rattributes, input)
}
const html = '<input type="email"
placeholder="hello@mjavascript.com" />'
console.log(...parseAttributes(html))
// [
//   ['type', 'email']
//   ['placeholder', 'hello@mjavascript.com']
// ]
```

然而，每次调用 `RegExp#exec` 时，`rattributes` 都会改变其 `lastIndex` 属性，这就是它在最后一次匹配后跟踪位置的方式。当没有匹配时，`lastIndex` 会重新设置为 `0`。如果我们不一步遍历所有可能匹配的输入，那么会产生一个问题，即 `lastIndex` 会重置为 `0`，然后我们在第二个输入上使用正则表达式，这会获得意料之外的结果。

虽然我们的 `matchAll` 实现看起来不会因为遍历所有的匹配而成为受害者，但手动迭代生成器是可能的。这意味着，如果重用相同的正则表达式，那我们就会遇到麻烦，如下所示。注意第二个匹配器应该如何报告 `['type', 'text']`，而不是从比 `0` 索引更远的索引开始，甚至将 `'placeholder'` 关键字误报为 `'laceholder'`。

```
const rattributes = /(\w+)="([^"]+)"\s/ig
const email = '<input type="email"
placeholder="hello@mjavascript.com" />'
const emailMatcher = matchAll(rattributes, email)
const address = '<input type="text"
placeholder="Enter your business address" />'
```

```
const addressMatcher = matchAll(rattributes, address)
console.log(emailMatcher.next().value)
// <- ['type', 'email']
console.log(addressMatcher.next().value)
// <- ['laceholder', 'Enter your business address']
```

其中一种解决方案是改变 `matchAll`，这样一来，当我们回到消费者代码时，`lastIndex` 始终为 `0`，同时保持 `lastIndex` 内部跟踪，以便我们可以在序列的每个步骤中找到离开的地方。

确实，这将解决我们所遇到的问题，如下所示。因此，最好可以避免使用可复用的全局正则表达式，这样我们就不用担心在每次使用后重置 `lastIndex` 了。

```
function* matchAll(regex, input) {
  let lastIndex = 0
  while (true) {
    regex.lastIndex = lastIndex
    const match = regex.exec(input)
    if (match === null) {
      break
    }
    lastIndex = regex.lastIndex
    regex.lastIndex = 0
    const [ , ...captures] = match
    yield captures
  }
}
const rattributes = /(\w+)="([^"]+)"\s/ig
const email = '<input type="email"
placeholder="hello@mjavascript.com" />'
const emailMatcher = matchAll(rattributes, email)
const address = '<input type="text"
placeholder="Enter your business address" />'
const addressMatcher = matchAll(rattributes, address)
console.log(emailMatcher.next().value)
// <- ['type', 'email']
console.log(addressMatcher.next().value)
// <- ['type', 'text']
console.log(emailMatcher.next().value)
// <- ['placeholder', 'hello@mjavascript.com']
console.log(addressMatcher.next().value)
// <- ['placeholder', 'Enter your business address']
```

`String#matchAll` 提议[13]（阶段 1）为字符串原型引入了一种新的方法,除了返回的迭代器是一系列 `match` 对象而不只是前面示例中的 `captures`，其行为与 `matchAll` 实现类似。注意，`String#matchAll` 序列包含整个 `match` 对象，而不只是编号捕获。这意味着我们可以通过 `match.groups` 访问命名捕获序列中的每个 `match`。

```
const rattributes = /(?<key>\w+)="(?<value>[^"]+)"\s/igu
const email = '<input type="email"
```

注 13：参见 `String#matchAll` 提案文档（https://github.com/tc39/proposal-string-matchall）。

```
placeholder="hello@mjavascript.com" />'
for (const match of email.matchAll(rattributes)) {
  const { groups: { key, value } } = match
  console.log(`${ key }: ${ value }`)
}
// <- type: email
// <- placeholder: hello@mjavascript.com
```

7.5 Array

多年以来，在谈及数组时，Underscore 和 Lodash 这样的库都高调抱怨其缺失的特性。后来，ES5 为数组添加了很多功能方法：`Array#filter`、`Array#map`、`Array#reduce`、`Array#reduceRight`、`Array#forEach`、`Array#some` 和 `Array#every`。

现在 ES6 又新增了一些有助于操作、填充和过滤数组的方法。

7.5.1 Array.from

在 ES6 问世前，JavaScript 开发人员通常需要通过一个函数将参数转化为数组。

```
function cast() {
  return Array.prototype.slice.call(arguments)
}
cast('a', 'b')
// <- ['a', 'b']
```

学习了第 2 章中的剩余参数和扩展运算符后，我们可以用几种简洁的方式来实现这个操作。扩展运算符可以使用迭代器协议在任意对象中生成一系列值，从而实现上述操作。但使用扩展运算符的缺点是，我们想要用扩展进行投射的对象必须通过实现 `Symbol.iterator` 来遵循迭代器协议。好在 ES6 中的 `arguments` 实现了迭代器协议。

```
function cast() {
  return [...arguments]
}
cast('a', 'b')
// <- ['a', 'b']
```

这种特殊情况下使用剩余参数更好，因为它不涉及 `arguments` 对象，也不会在函数体中添加任何逻辑。

```
function cast(...params) {
  return params
}
cast('a', 'b')
// <- ['a', 'b']
```

或许你想要通过扩展运算符来扩展 `NodeList` DOM 元素集合，如通过 `document.query-`

SelectorAll 选定的元素。这在使用 Array#map 或 Array#filter 这样的本地数组方法时会很方便。ES6 重新定义了迭代器协议，新的 DOM 标准将 NodeList 升级为可迭代的，从而使得上述操作成为可能。

```
[...document.querySelectorAll('div')]
// <- [<div>, <div>, <div>, ...]
```

试图通过扩展运算符来扩展 jQuery 集合时会发生什么呢？如果你使用的是实现迭代器协议的新版 jQuery，扩展 jQuery 对象将会生效，否则就会报告异常。

```
[...$('div')]
// <- [<div>, <div>, <div>, ...]
```

新的 Array.from 方法有些不同。它不会依赖迭代器协议从对象中提取值，与扩展运算符不同的是，它支持任何类数组对象。以下代码段可用于任何版本的 jQuery。

```
Array.from($('div'))
// <- [<div>, <div>, <div>, ...]
```

无论是 Array.from 还是扩展运算符，都无法做到选择一个开始索引。假设你想要获取第一个 <div> 元素后的所有 <div> 元素，那么可以使用 Array#slice。

```
[].slice.call(document.querySelectorAll('div'), 1)
```

当然，你也可以在进行转换后再使用 Array#slice。这种做法比前面示例的可读性更高，因为我们可以很明确地看到想要分割数组的位置。

```
Array.from(document.querySelectorAll('div')).slice(1)
```

Array.from 有 3 个参数，但只有 input 这个参数是必需的：

- input，你想要扩展的类数组对象或可迭代对象；
- map，应用于 input 的每个元素的映射函数；
- context，调用 map 时的 this 绑定。

虽然 Array.from 不能进行对象的分割，但可以对对象进行逻辑处理。map 函数可以有效地将值映射到其他值上，在调用 Array.from 的同时进行映射处理，并生成想要的数组。

```
function typesOf() {
  return Array.from(arguments, value => typeof value)
}
typesOf(null, [], NaN)
// <- ['object', 'object', 'number']
```

注意，在处理 arguments 的特殊情况时，还可以将剩余参数和 Array#map 搭配使用。同前面的 Array#slice 示例相同，本着准确清晰的原则，以下示例展示的处理更为合适。

```
function typesOf(...all) {
  return all.map(value => typeof value)
}
typesOf(null, [], NaN)
// <- ['object', 'object', 'number']
```

在处理类似数组的对象时，如果没有实现 `Symbol.iterator`，使用 `Array.from` 则是有意义的。

```
const apple = {
  type: 'fruit',
  name: 'Apple',
  amount: 3
}
const onion = {
  type: 'vegetable',
  name: 'Onion',
  amount: 1
}
const groceries = {
  0: apple,
  1: onion,
  length: 2
}
Array.from(groceries)
// <- [apple, onion]
Array.from(groceries, grocery => grocery.type)
// <- ['fruit', 'vegetable']
```

7.5.2 Array.of

`Array.of` 方法与前面用过的 `cast` 函数完全相同。以下代码展示了如何编写一个 `Array.of` 的 ponyfill。

```
function arrayOf(...items) {
  return items
}
```

`Array` 构造函数有两个重载参数：可以直接传入数组元素的 `...items`，以及传入数组具体长度的 `length`。`Array.of` 也可以生成一个新的数组，只是该数组不支持传入数组长度。在以下代码中，由于使用单参数 `length` 重载构造函数，你会发现 `new Array` 的奇怪之处。或许你会对浏览器控制台中的 `undefined x ${ count }` 符号感到困惑，这说明该数组中存在空位，这种数组也称为**稀疏数组**。

```
new Array() // <- []
new Array(undefined) // <- [undefined]
new Array(1) // <- [undefined x 1]
new Array(3) // <- [undefined x 3]
new Array('3') // <- ['3']
new Array(1, 2) // <- [1, 2]
new Array(-1, -2) // <- [-1, -2]
new Array(-1) // <- RangeError: Invalid array length
```

相反，由于不支持传入数组长度，Array.of 的行为更为可控。就稳定性而言，这种构建数组的方式更为理想。

```
console.log(Array.of()) // <- []
console.log(Array.of(undefined)) // <- [undefined]
console.log(Array.of(1)) // <- [1]
console.log(Array.of(3)) // <- [3]
console.log(Array.of('3')) // <- ['3']
console.log(Array.of(1, 2)) // <- [1, 2]
console.log(Array.of(-1, -2)) // <- [-1, -2]
console.log(Array.of(-1)) // <- [-1]
```

7.5.3 Array#copyWithin

我们从 Array#copyWithin 签名开始着手。

```
Array.prototype.copyWithin(target, start = 0, end = this.length)
```

Array#copyWithin 方法将数组中的若干成员复制到该数组的 target 位置，被复制的成员是该数组中 [start, end) 这一区间内的元素。Array#copyWithin 方法返回数组本身。

我们来看一个简单示例，思考以下示例中的 items 数组。

```
const items = [1, 2, 3, , , , , , , ]
// <- [1, 2, 3, undefined x 7]
```

以下示例中的函数接收一个 items 数组，并决定粘贴到的目标位置是索引 6（从 0 开始），被复制的元素是原数组的索引 1~3 之间的元素（包含索引 1，但不包含索引 3）。

```
const items = [1, 2, 3, , , , , , , ]
items.copyWithin(6, 1, 3)
// <- [1, 2, 3, undefined × 3, 2, 3, undefined × 2]
```

如果对你来说迅速理解 Array#copyWithin 很难，那我们来看看分解动作。

根据定义，被复制的成员在 [start, end) 范围内，我们可以调用 Array#slice 来得到这些预计粘贴在 target 位置中的成员。我们可以使用 .slice 来获取副本。

```
const items = [1, 2, 3, , , , , , , ]
const copy = items.slice(1, 3)
// <- [2, 3]
```

粘贴操作可以视为 Array#splice 的高级用法。在以下代码中，我们将要粘贴的位置传递给 splice，同时删除该位置上与粘贴成员数量相同的数组成员，并插入粘贴的成员。注意，我们使用的是扩展运算符，因此元素是通过 .splice 单独插入的，而不是作为数组插入。

```
const items = [1, 2, 3, , , , , , , ]
const copy = items.slice(1, 3)
```

```
// <- [2, 3]
items.splice(6, 3 - 1, ...copy)
console.log(items)
// <- [1, 2, 3, undefined × 3, 2, 3, undefined × 2]
```

现在我们更好地理解了 Array#copyWithin 的内部运作，我们可以简化操作来实现自定义的 copyWithin 函数，如下所示。

```
function copyWithin(
  items,
  target,
  start = 0,
  end = items.length
) {
  const copy = items.slice(start, end)
  const removed = end - start
  items.splice(target, removed, ...copy)
  return items
}
```

使用自定义的 copyWithin 也能完美实现 Array#copyWithin 的功能。

```
copyWithin([1, 2, 3, , , , , , , , ], 6, 1, 3)
// <- [1, 2, 3, undefined × 3, 2, 3, undefined × 2]
```

7.5.4　Array#fill

这是一种方便且实用的方法，可以用提供的 value 替换数组中的所有成员。注意，在稀疏数组中，这个值将填充空位并替换现有成员。

```
['a', 'b', 'c'].fill('x') // <- ['x', 'x', 'x']
new Array(3).fill('x') // <- ['x', 'x', 'x']
```

你也可以指定起始索引和结束索引，只有指定位置的成员才会被填充，如下所示。

```
['a', 'b', 'c', , ,].fill('x', 2)
// <- ['a', 'b', 'x', 'x', 'x']
new Array(5).fill('x', 0, 1)
// <- ['x', undefined x 4]
```

提供的 value 可以是任意值，并不局限于基本类型值。

```
new Array(3).fill({})
// <- [{}, {}, {}]
```

但是，你不能用带有 index 参数或类似参数的映射方法来填充数组。

```
const map = i => i * 2
new Array(3).fill(map)
// <- [map, map, map]
```

7.5.5 Array#find和Array#findIndex

Array#find 方法为数组中的每个成员执行 callback 函数，直到有成员返回 true 为止，然后返回该成员。该方法遵循的签名是 (callback(item, i, array), context)，使用方式同 Array#map、Array#filter 等方法相同。你可以认为 Array#find 是一个返回匹配元素（而不只是 true）的 Array#some。

```
['a', 'b', 'c', 'd', 'e'].find(item => item === 'c')
// <- 'c'
['a', 'b', 'c', 'd', 'e'].find((item, i) => i === 0)
// <- 'a'
['a', 'b', 'c', 'd', 'e'].find(item => item === 'z')
// <- undefined
```

还有一种 Array#findIndex 方法，它使用相同的签名。不同的是，Array.findIndex 返回匹配元素的索引，而不是一个布尔值或元素本身。当没有匹配成功时，Array.findIndex 的返回值为 -1，如下所示。

```
['a', 'b', 'c', 'd', 'e'].findIndex(item => item === 'c')
// <- 2
['a', 'b', 'c', 'd', 'e'].findIndex((item, i) => i === 0)
// <- 0
['a', 'b', 'c', 'd', 'e'].findIndex(item => item === 'z')
// <- -1
```

7.5.6 Array#keys

Array#keys 返回一个迭代器，该迭代器产生一个保存数组键的序列。返回值是一个迭代器，意味着你可以使用 for..of、扩展运算符或手动调用 .next() 来迭代它。

```
['a', 'b', 'c', 'd'].keys()
// <- ArrayIterator {}
```

以下是 for..of 的一个用例。

```
for (const key of ['a', 'b', 'c', 'd'].keys()) {
  console.log(key)
  // <- 0
  // <- 1
  // <- 2
  // <- 3
}
```

与 Object.keys 以及大多数迭代数组的方法不同，该序列不会忽略数组中的空位。

```
Object.keys(new Array(4))
// <- []
[...new Array(4).keys()]
// <- [0, 1, 2, 3]
```

接下来轮到值了。

7.5.7 Array#values

Array#values 和 Array#keys() 相同，但返回的迭代器是一系列值而不是键。实际上，你可能会想要迭代数组本身，但有时获得一个迭代器可能更为方便一些。

```
['a', 'b', 'c', 'd'].values()
// <- ArrayIterator {}
```

你可以使用 for..of 或其他任何方法（如扩展运算符）来提取可迭代的序列。以下示例在数组的 .values() 上使用扩展运算符来创建该数组的副本。

```
[...['a', 'b', 'c', 'd'].values()]
// <- ['a', 'b', 'c', 'd']
```

注意，省略 .values() 方法调用仍然会生成数组的副本：序列被迭代并扩展在新数组中。

7.5.8 Array#entries

Array#entries 与前面的两个方法类似，但其返回值是含有一系列键 / 值对的迭代器。

```
['a', 'b', 'c', 'd'].entries()
// <- ArrayIterator {}
```

序列中的每个成员是键和值组成的二维数组。

```
[...['a', 'b', 'c', 'd'].entries()]
// <- [[0, 'a'], [1, 'b'], [2, 'c'], [3, 'd']]
```

现在只剩下最后一个方法了！

7.5.9 Array.prototype[Symbol.iterator]

这个方法与 Array#values 完全相同。

```
const list = ['a', 'b', 'c', 'd']
list[Symbol.iterator] === list.values
// <- true
[...list[Symbol.iterator]()]
// <- ['a', 'b', 'c', 'd']
```

以下示例结合使用扩展运算符、数组和 Symbol.iterator 进行数组值的迭代。你能读懂这段代码吗？

```
[...['a', 'b', 'c', 'd'][Symbol.iterator]()]
// <- ['a', 'b', 'c', 'd']
```

我们用分解的思想来分析一下这段代码。首先，有这样一个数组。

```
['a', 'b', 'c', 'd']
// <- ['a', 'b', 'c', 'd']
```

接着，我们得到一个迭代器。

```
['a', 'b', 'c', 'd'][Symbol.iterator]()
// <- ArrayIterator {}
```

最后，我们将迭代器扩展在一个新的数组上，以创建一个副本。

```
[...['a', 'b', 'c', 'd'][Symbol.iterator]()]
// <- ['a', 'b', 'c', 'd']
```

第 8 章

JavaScript模块

近几年涌现了很多分而治之的代码管理方案。长久以来，我们一直使用所谓的模块模式，即将代码包装到自调用的函数表达式中。我们必须自己管理脚本的次序，以确保每个模块都在自己的依赖之后加载。

后来，RequireJS 库出现了。它提供了以编程方式定义每个模块依赖的机制，有了依赖图后，我们就不必再操心脚本的加载次序了。RequireJS 接受一个字符串数组，以明确依赖，同时将模块包装为一个函数调用，而这个函数接受前面的依赖作为参数。其实很多其他库都支持类似的功能，只不过 API 不大一样。

还有其他的复杂依赖管理机制，如 AngularJS 中的依赖注入机制。该机制需要我们用函数定义命名组件，并相应地指定其他命名组件依赖。AngularJS 会替我们管理依赖注入的加载，我们要做的只是命名组件和指定依赖。

取代 RequireJS 的是 CommonJS，随着 Node.js 的火爆，CommonJS 迅速走红。本章将先介绍 CommonJS，毕竟它在今天的应用还相当普遍。接下来本章将介绍 ES6 为 JavaScript 引入的原生模块系统。最后，本章将探讨 CommonJS 与原生 JavaScript 模块（即人们常说的 ECMAScript 模块）的互用性。

8.1 CommonJS

CommonJS 与其他模块化方案的不同之处是，它将每个文件都看成一个模块，其他方案则以编程方式声明模块。CommonJS 模块拥有隐含的局部作用域，全局作用域必须通过 `global` 显式地被访问。CommonJS 模块导入依赖及导出供外部调用的接口都是动态的，导入依赖是通

过 `require` 函数调用实现的。这个函数调用是同步的，会返回所请求模块暴露的接口。

不看代码很难说清一种模块的定义语法。以下代码展示了一个可重用的 CommonJS 模块文件。其中的 `has` 和 `union` 函数都位于模块的局部作用域内。在此之上，我们再将 `union` 赋给 `module.exports`，它就成了这个模块的公共 API。

```
function has(list, item) {
  return list.includes(item)
}
function union(list, item) {
  if (has(list, item)) {
    return list
  }
  return [...list, item]
}
module.exports = union
```

假设将以上文件保存为 union.js，那我们就可以在另一个 CommonJS 模块中调用它。比如，另一个文件是 app.js，为了调用 union.js，需要给 `require` 传入一个相对路径。

```
const union = require('./union.js')
console.log(union([1, 2], 3))
// <- [1, 2, 3]
console.log(union([1, 2], 2))
// <- [1, 2]
```



如果扩展名是 .js 或 .json，则可以省略，但不鼓励这么做。

虽然扩展名对 `require` 语句来说是可选的，在使用 `node` CLI 时最好还是养成添加扩展名的习惯。浏览器对 ESM 的实现不允许省略扩展名，否则就会导致多一次到服务器的往返才能找到作为 HTTP 资源的 JavaScript 模块的正确端点。

在 Node.js 环境下，我们可以通过 CLI 运行 app.js，如下所示。

```
» node app.js
# [1, 2, 3]
# [1, 2]
```



安装 Node.js 后，就可以在终端命令行中使用 `node` 程序。

正如其他 JavaScript 函数那样，CommonJS 中的 `require` 函数也可以被动态调用。我们可以利用这一特性实现通过一个接口动态获取不同的模块。举个例子，假设有一个模板目录，其中包含几个视图模板文件，每个文件都导出一个函数。这些函数都接受一个模型参数，然后返回一个 HTML 字符串。

通过读取 `model` 对象的属性，以下代码中展示的模板函数构造并返回购物车中的一个商品。

```
// views/item.js
module.exports = model => `<li>
  <span>${ model.amount }</span>
  <span>x </span>
  <span>${ model.name }</span>
</li>`
```

应用模块可以基于这个 item.js 提供的视图模板打印一个 `<li>` 元素。

```
// app.js
const renderItem = require('./views/item.js')
const html = renderItem({
  name: 'Banana bread',
  amount: 3
})
console.log(html)
```

图 8-1 展示了这个小应用的执行结果。

```
bevacqua@MacBook-Pro: ~/dev/practical-modern-javascript/code/ch08/ex01-cjs-grocery-item

master  ~/dev/practical-modern-javascript/code/ch08/ex01-cjs-grocery-item
» cat app.js
const renderItem = require('./views/item')
const html = renderItem({
  name: 'Banana bread',
  amount: 3
})
console.log(html)
master  ~/dev/practical-modern-javascript/code/ch08/ex01-cjs-grocery-item
» node app
<li>
  <span>3</span>
  <span>x </span>
  <span>Banana bread</span>
</li>
master  ~/dev/practical-modern-javascript/code/ch08/ex01-cjs-grocery-item
»
```

图 8-1：将模型渲染为 HTML 就是向模板字面量中插值而已

接下来的这个模板用于渲染购物车中的所有商品。它接受一个商品数组，并重用前面的 item.js 模板来渲染每件商品。

```
// views/list.js
const renderItem = require('./item.js')

module.exports = model => `<ul>
  ${ model.map(renderItem).join('\n') }
</ul>`
```

我们可以像前面那样使用 list.js 模板。但要注意，传给它的模型必须是一个商品数组，而非单个商品对象。

```
// app.js
const renderList = require('./views/list.js')
const html = renderList([{
  name: 'Banana bread',
  amount: 3
}, {
  name: 'Chocolate chip muffin',
  amount: 2
}])
console.log(html)
```

图 8-2 展示了小应用的当前状况。

```
bevacqua@MacBook-Pro: ~/dev/practical-modern-javascript/code/ch08/ex02-cjs-grocery-list

 master  ~/dev/practical-modern-javascript/code/ch08/ex02-cjs-grocery-list
» cat app.js
const renderList = require('./views/list')
const html = renderList([{
  name: 'Banana bread',
  amount: 3
}, {
  name: 'Chocolate chip muffin',
  amount: 2
}])
console.log(html)
 master  ~/dev/practical-modern-javascript/code/ch08/ex02-cjs-grocery-list
» node app
<ul>
  <li>
    <span>3</span>
    <span>x </span>
    <span>Banana bread</span>
  </li>
  <li>
    <span>2</span>
    <span>x </span>
    <span>Chocolate chip muffin</span>
  </li>
</ul>
 master  ~/dev/practical-modern-javascript/code/ch08/ex02-cjs-grocery-list
»
```

图 8-2：基于模板字面量的复合模板同样是信手拈来

到目前为止，这个示例只写了几个小模块，每个模块只基于传入的模型对象和视图模板生成一种 HTML 视图。简单的 API 便于重用，因此我们才能轻松地将模型映射到 item.js 模板函数，以渲染出多个商品，最后再通过换行符将它们连接起来。

既然两个视图的 API 相似，都是接受一个模型并返回一段 HTML 字符串，那么就可以抽象一下。如果想要一个 `render` 函数能够渲染任何模板，那么可以借助 `require` 的动态特性来轻松实现。以下示例的核心是构建指向模板模块的路径。与前面代码的重要不同点是，`require` 调用并没有出现在模块代码的顶部。对 `require` 的调用可以出现在任何地方，甚至可以嵌套在其他函数中。

```
// render.js
module.exports = function render(template, model) {
  return require(`./views/${ template }`.js)(model)
}
```

有了这样的 API 后，就不用操心调用 `require` 时传入的视图模板路径是否正确了，因为 render.js 模块会正确拼接出路径。要想渲染模板，只要传入模板的名字和该模板所需要的模型即可，如以下代码和图 8-3 所示。

```
// app.js
const render = require('./render.js')
console.log(render('item', {
  name: 'Banana bread',
  amount: 1
}))
console.log(render('list', [{
  name: 'Apple pie',
  amount: 2
}, {
  name: 'Roasted almond',
  amount: 25
}]))
```

```
bevacqua@MacBook-Pro: ~/dev/practical-modern-javascript/code/ch08/ex03-cjs-dynamic-render

 master  ~/dev/practical-modern-javascript/code/ch08/ex03-cjs-dynamic-render
» cat app.js
const render = require('./render')
console.log(render('item', {
  name: 'Banana bread',
  amount: 1
}))
console.log(render('list', [{
  name: 'Apple pie',
  amount: 2
}, {
  name: 'Roasted almond',
  amount: 25
}]))
 master  ~/dev/practical-modern-javascript/code/ch08/ex03-cjs-dynamic-render
» node app
<li>
  <span>1</span>
  <span>x </span>
  <span>Banana bread</span>
</li>
<ul>
  <li>
    <span>2</span>
    <span>x </span>
    <span>Apple pie</span>
  </li>
  <li>
    <span>25</span>
    <span>x </span>
    <span>Roasted almond</span>
  </li>
</ul>
 master  ~/dev/practical-modern-javascript/code/ch08/ex03-cjs-dynamic-render
»
```

图 8-3：模板字面量使得创建 HTML 渲染应用变得易如反掌

接下来，我们会看到 ES6 模块从某种程度上受到了 CommonJS 的影响。接下来几节会讨论 `export` 和 `import` 语句，以及 ESM 与 CommonJS 有哪些相通之处。

8.2 JavaScript模块

在前面介绍 CommonJS 模块时，我们已经知道其 API 简单却非常灵活、强大。ES6 模块的 API 甚至比它还要简单，虽然灵活性稍微差了点，但几乎是一样强大的。

8.2.1 严格模式

在 ES6 模块系统中，严格模式默认是打开的。严格模式是一个特性[1]，用于拒绝 JavaScript 中那些不好的特性，并让很多静默错误变成异常，从而被抛出。在拒绝这些特性的基础上，编译器可以启用优化策略，让 JavaScript 运行时更快、更安全。

- 变量必须被声明。
- 函数参数的名字必须是唯一的。
- 禁止使用 `with` 语句。
- 为只读属性赋值会导致抛出错误。
- `00740` 这样的八进制数是语法错误。
- 用 `delete` 删除不可删除的属性会抛出错误。
- `delete prop` 是语法错误，`delete global.prop` 才是正确的。
- `eval` 不会为周围的作用域引入新变量。
- `eval` 和 `arguments` 不能被绑定或赋值。
- `arguments` 不会神奇地同步方法参数的变化。
- 不再支持 `arguments.callee`，访问它会抛出 `TypeError`。
- 不再支持 `arguments.caller`，访问它会抛出 `TypeError`。
- 方法调用中作为 `this` 传递的上下文不会被“装箱”为 `Object`。
- 不能再使用 `fn.caller` 和 `fn.arguments` 来访问 JavaScript 栈。
- 保留字（如 `protected`、`static`、`interface` 等）不能被绑定。

接下来我们将深入探讨一下 `export` 语句。

8.2.2 export语句

在 CommonJS 模块中，要导出的值必须赋给 `module.exports`。可以导出的内容包括任意类型的值、对象、数组、函数，如下所示。

注 1：建议阅读一下有关 MDN 上严格模式的完整介绍（https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Strict_mode）。

```
module.exports = 'hello'

module.exports = { hello: 'world' }

module.exports = ['hello', 'world']

module.exports = function hello() {}
```

作为文件，ES6 模块是通过 `export` 语句暴露 API 的。ESM 中的声明只在局部作用域中有效，这一点与 CommonJS 相同。模块中声明的任何变量都必须作为该模块的 API 导入，并且在想要使用它们的模块中导入才能访问。

1. 导出默认绑定

将前面 CommonJS 代码中的 `module.exports =` 替换成 `export default` 即可在 ESM 中实现相同的效果。

```
export default 'hello'

export default { hello: 'world' }

export default ['hello', 'world']

export default function hello() {}
```

在 CommonJS 中，我们可以为 `module.exports` 动态赋值。

```
function initialize() {
  module.exports = 'hello!'
}
initialize()
```

相较于 CommonJS，ESM 中的 `export` 语句只能出现在模块顶级。`export` 语句“只能出现在顶级”是一个很好的限制，因为根据方法调用来动态定义并暴露 API 并不是非常必要。这一限制还有助于编译器及静态分析工具解析 ES6 模块。

```
function initialize() {
  export default 'hello!' // SyntaxError
}
initialize()
```

除了 `export default` 语句，ESM 还支持其他暴露 API 的方式。

2. 命名导出

在 CommonJS 中，如果想要暴露多个值，不一定需要导出一个包含这些值的对象。我们可以将这些值赋给隐含的 `module.exports` 对象。这样导出的仍然只是一个绑定，其中包含 `module.exports` 对象最终持有的所有属性。也就是说，虽然以下示例看似导出了两个值，但实际上它们都是最终导出对象的属性。

```
module.exports.counter = 0
module.exports.count = () => module.exports.counter++
```

在 ESM 中，我们可以通过命名导出语法复现这一行为。相较于 CommonJS 为隐含的 `module.exports` 对象添加属性，ES6 是直接声明要导出的绑定，如下所示。

```
export let counter = 0
export const count = () => counter++
```

注意，前面的代码不能将变量声明提取为独立的语句，然后再作为命名导出传给 `export`，否则会导致语法错误。

```
let counter = 0
const count = () => counter++
export counter // SyntaxError
export count
```

ESM 这样严格限制模块中声明的语法是为了方便静态分析，但代价是损失一些灵活性。要想提高灵活性，就必然会提高复杂性，这也是 ESM 不提供灵活接口的正当理由。

3. 导出列表

ES6 模块支持导出顶级命名成员的列表，如下所示。这种导出列表的语法很容易解析，同时也就前面提出的问题给出了一个解决方案。

```
let counter = 0
const count = () => counter++
export { counter, count }
```

要想重命名导出的绑定，可以使用别名语法 `export { count as increment}`。这样我们就可以将局部作用域中的绑定 `count` 以别名 `increment` 提供给外部，如下所示。

```
let counter = 0
const count = () => counter++
export { counter, count as increment }
```

最后，使用命名成员列表语法时还可以指定默认导出。以下代码使用 `as default` 在导出多个命名成员的同时定义了模块的默认导出。

```
let counter = 0
const count = () => counter++
export { counter as default, count as increment }
```

虽然以下代码长了点，但功能与上段代码相同。

```
let counter = 0
const count = () => counter++
export default counter
export { count as increment }
```

需要特别注意的是，我们导出的是绑定，而不只是值。

4. 绑定，而不是值

ES6 模块导出绑定，而不是值或引用。这意味着以下示例中的模块导出的 `fungible` 会绑定到这个模块的 `fungible` 变量，其值会随 `fungible` 变量的变化而变化。被其他模块引用后再改变公共接口可能会导致困惑，但这一机制在某些情况下确实也很有用。

在以下代码中，模块导出的 `fungible` 一开始绑定的是一个对象，5 秒后又改成了一个数组。

```
export let fungible = { name: 'bound' }
setTimeout(() => fungible = [0, 1, 2], 5000)
```

使用这个 API 的模块在 5 秒后也能看到 `fungible` 值的变化。以下示例每隔 2 秒会打印一次引入的绑定。

```
import { fungible } from './fungible.js'

console.log(fungible) // <- { name: 'bound' }
setInterval(() => console.log(fungible), 2000)
// <- { name: 'bound' }
// <- { name: 'bound' }
// <- [0, 1, 2]
// <- [0, 1, 2]
// <- [0, 1, 2]
```

这种行为特别适合计数器和标记，但除非用途明确，最好不要使用。毕竟从使用者的角度来看，API 接口不确定很难理解。

JavaScript 的模块系统还提供了一个 `export..from` 语法，用于暴露其他模块的接口。

5. 导出另一个模块

向 `export` 添加一个 `from` 子句就可以导出另一个模块的命名导出。此时绑定不会导入到当前模块的作用域。换句话说，当前模块只是传递另一个模块的绑定，并不能直接访问该绑定。

```
export { increment } from './counter.js'
increment()
// ReferenceError: increment is not defined
```

在传递绑定时，我们可以为命名导出起个别名。如果以下示例中的模块被命名为 `aliased`，那么调用者可以通过 `import { add } from 'aliased.js'` 取得 `counter` 模块中的 `increment` 的绑定。

```
export { increment as add } from './counter.js'
```

ESM 也支持用通配符导出另一个模块中的所有命名导出，如下所示。但要注意，此时不会导出 `counter` 模块中的默认绑定。

```
export * from './counter.js'
```

要想导出另一个模块的 default 绑定，必须使用导出列表语法来添加别名。

```
export { default as counter } from './counter.js'
```

我们将 ES6 模块暴露 API 的所有语法都过了一遍。接下来我们将探讨一下 import 语句，看看如何通过它使用其他模块。

8.2.3 import语句

我们可以用 import 语句在一个模块中加载另一个模块。加载模块的语法因实现而不同，也就是说，规范并未就此给出描述。如今，我们可以编写兼容 ES6 规范的代码，而一些聪明的人已经找到了在浏览器中处理模块加载的办法。

Babel 这样的编译器可以基于 CommonJS 等模块系统拼接模块。这意味着 Babel 中的 import 语句与 CommonJS 中的 require 具有相同的语义。

假设模块 ./counter.js 包含以下代码。

```
let counter = 0
const increment = () => counter++
const decrement = () => counter--
export { counter as default, increment, decrement }
```

以下这行代码可以将 counter 模块加载到我们的 app 模块中。这行代码不会在 app 模块的作用域中创建任何变量。但是这会导致 counter 模块中的所有顶级代码执行，包括该模块自己的 import 语句。

```
import './counter.js'
```

与 export 语句类似，import 语句也只允许出现在模块代码的顶级。这一限制有助于编译器简化自己的模块加载逻辑，同时有助于其他静态分析工具解析你的代码。

1. 导入默认导出

CommonJS 模块通过 require 语句导入其他模块。如果需要引入某个模块的默认导出，只要将该语句的结果赋给一个变量即可。

```
const counter = require('./counter.js')
```

要想导入 ES6 模块导出的默认绑定，就必须给它起个名字。但语法和语义与声明变量时有些不同，因为这里是导入绑定，而不只是将值赋给一个变量。这个区别有助于静态分析工具和编译器更轻松地解析我们的代码。

```
import counter from './counter.js'
console.log(counter)
// <- 0
```

除了默认导出，我们也可以导入命名导出并给它们起别名。

2. 导入命名导出

以下代码展示了如何从 counter 模块导入 increment 方法。导入命名导出的语法是一对花括号，这让我们联想到了解构赋值。

```
import { increment } from './counter.js'
```

为了导入多个绑定，绑定之间以逗号分隔。

```
import { increment, decrement } from './counter.js'
```

这里的语法和语义与解构有些不同。比如，解构通过冒号创建别名，而 import 语句则使用 as 关键字，照搬了 export 语句的语法。以下代码在导入 increment 方法时将其重命名为 add。

```
import { increment as add } from './counter.js'
```

以逗号作为分隔符，我们可以同时导入默认导出和命名导出。

```
import counter, { increment } from './counter.js'
```

我们也可以给 default 绑定命名，此时需要一个别名。

```
import { default as counter, increment } from './counter.js'
```

以下的代码示例展示了 ESM 与 CommonJS 导入在语义上的区别。记住，ESM 中导出和导入的是绑定，而不是引用。为方便理解，你可以将以下代码中的绑定 counter 想象为一个属性的获取方法（getter），它可以访问 counter 模块内部并返回其局部变量 counter 的值。

```
import counter, { increment } from './counter.js'
console.log(counter) // <- 0
increment()
console.log(counter) // <- 1
increment()
console.log(counter) // <- 2
```

最后，我们将探讨命名空间导入。

3. 通配符导入语句

我们可以用通配符导入一个模块的命名空间对象。相较于导入命名导出或默认导出，这样可以一次性导入所有导出。注意，星号 * 后面必须紧跟别名，导入的所有绑定都在这个别名名下。如果存在 default 导出，那么它也会被放到这个命名空间绑定之下。

```
import * as counter from './counter.js'
counter.increment()
counter.increment()
console.log(counter.default) // <- 2
```

8.2.4 动态import()

在撰写本书时，有人提出了有关动态 `import()` 表达式的提案（阶段 3）。与 `import` 语句的静态分析和链接不同，`import()` 在运行时加载模块，并在获取、解析并运行请求的模块及其所有依赖后，返回一个包含该模块命名空间对象的 Promise。

与 `import` 语句类似，此时的模块说明符可以是任意字符串。但与 `import` 只允许静态的字符串字面量作为模块说明符不同，`import()` 的模块说明符可以是模板字面量或任何能生成模块说明符字符串的有效 JavaScript 表达式。

假设我们正在国际化某个应用，需要根据用户代理的语言偏好加载相应的语言包。我们可以先导入 `localizationService`，然后再通过 `import()` 及根据插入 `navigator.language` 的模板字面量构造的模块说明符来实现本地化数据的动态加载，如下所示。

```
import localizationService from './localizationService.js'
import(`./localizations/${ navigator.language }.json`)
  .then(module => localizationService.use(module))
```

注意，通常并不建议将代码写成以上那样，原因如下。

- 对静态分析不友好，因为静态分析是在构建时执行的，所以几乎不可能推断出 `${ navigator.language }` 这样插值的结果。
- JavaScript 打包工具也很难对其进行打包，结果很可能是应用加载完成后再异步加载这个模块。
- Rollup 等工具很难对其进行摇树优化，难以消除并未导入（因而永远不会用到）的模块代码，因此也就难以缩小包并提升性能。
- 不利于辅助检查模块导入语句中要导入的文件是否缺失的工具（如 `eslint-plugin-import`）发挥作用。

与 `import` 语句类似，规范也没有说明 `import()` 获取模块的方式，因此就要看宿主环境了。

但提案说明了模块被成功解决后，Promise 应该获取解析后的命名空间对象。同时该提案指出，如果发生错误导致模块加载失败，那么 Promise 应该被拒绝。

对于不那么重要的模块，我们可以在不阻塞页面加载的情况下进行异步加载，同时还可以在模块加载失败时恰当地处理，如下所示。

```
import('./vendor/jquery.js')
  .then($ => {
  // 使用jQuery
})
.catch(() => {
  // 加载jQuery失败
})
```

我们可以用 Promise.all 异步加载多个模块。以下示例同时导入了 3 个模块，然后在 .then 子句中直接用解构获取了对它们的引用。

```
const specifiers = [
  './vendor/jquery.js',
  './vendor/backbone.js',
  './lib/util.js'
]
Promise
  .all(specifiers.map(specifier => import(specifier)))
  .then(([$, backbone, util]) => {
    // 使用模块
  })
```

同样，我们可以用同步循环或 async/await 来加载模块，如下所示。

```
async function load() {
  const { map } = await import('./vendor/jquery.js')
  const $ = await import('./vendor/jquery.js')
  const response = await fetch('/cats')
  const cats = await response.json()
  $('<div>')
    .addClass('container cats')
    .html(map(cats, cat => cat.htmlSnippet))
    .appendTo(document.body)
}
load()
```

await import() 让动态导入模块看起来像静态的 import 语句。我们自己心里必须明白，这里其实是一个接一个地异步加载多个模块。

注意，虽然 import() 有点像函数，但语义与常规函数不同。这里 import 并非函数定义，不能进行扩展、不能给它添加属性，也不能对它使用解构语法。从这个意义上说，import() 更像是类构造器中的 super() 调用。

8.3 ES模块的实践考量

无论使用什么模块系统，我们都可以做到公开 API 并同时隐藏信息。这种完美的信息隐藏正是以前的开发者梦寐以求的特性。那时候，要想实现同样的功能，必须非常熟悉 JavaScript 的作用域规则，否则就得盲目地循环某种模式，如下所示。这个示例使用局部作用域的 calc 函数创建了一个 random 模块，该函数负责生成一个区间为 [0, n) 的随机数，而公共 API 中包含 range 方法，该方法可以计算一个 [min, max] 范围内的随机数。

```
const random = (function() {
  const calc = n => Math.floor(Math.random() * n)
  const range = (max = 1, min = 0) => calc(max + 1 - min) + min
  return { range }
})()
```

比较以上代码与以下名为 `random` 的 ESM 模块中的代码。你会发现，立即调用函数表达式（IIFE，immediately invoked function expression）的包装不见了，模块的名字也不见了。这里模块的名字已经变成了文件名。我们又回到了以前在 HTML 的 `<script>` 标签内编写原始 JavaScript 代码的日子。

```
const calc = n => Math.floor(Math.random() * n)
const range = (max = 1, min = 0) => calc(max + 1 - min) + min
export { range }
```

虽然没有用 IIFE 来包装代码的问题了，但如何定义、测试、说明和使用模块仍然需要认真思考。

决定模块中包含什么内容并不容易。需要考虑的因素非常多，以下列举了其中一部分。

- 过于复杂吗？
- 过大了吗？
- API 有没有明确的含义？
- API 有没有完善的文档？
- 为这个模块编写测试是否简单？
- 为这个模块增加新特性难不难？
- 删除模块中的特性困难吗？

相较于模块长度，复杂性是需要考量的首要指标。一个模块可能有几千行代码但很简单，比如将文件说明符映射为特定语言字符串的字典；模块也可能只有几十行代码却非常难以理解，比如涉及域名验证和其他业务逻辑规则的数据模型。我们可以将代码拆分成更小的模块以降低复杂性，每个模块只专注于解决问题的某一方面。只要不是过于复杂，大模块也不是什么大问题。

明确定义的 API 同时配有完善的文档也是优秀模块化设计的关键。模块的 API 应该聚焦，遵循信息隐藏原理。换句话说，只对模块使用者暴露必要的东西。通过隐藏模块的内部实现，即使模块代码缺乏注释和文档，或者将来再被修改，我们仍然可以从整体上保持接口简单，避免意外的调用出现。通过给公开的 API 编写完善的文档，即使这些文档是写在代码中的注释，抑或代码本身就可以自解释，我们可以降低模块使用者的认知门槛。

测试应该只针对模块的公开接口来编写，模块的内部实现应该看作无关紧要的实现细节。测试要覆盖模块公开接口的不同方面，只要 API 的输入和输出不变，对内部实现的修改就不应该影响测试覆盖率。

同样，为模块增加或减少特性的容易性也是需要考量的一个因素。

- 添加一个新特性有多难？
- 为实现某个逻辑是不是必须修改几个不同的模块？

- 这个流程是不是重复了很多次？或许我们可以将那些变化抽象到一个高层模块，以隐藏复杂性，也许这样做很大程度上只是引入了一个中间层，虽然有一些好处或改进，却导致代码更难理解了。
- 从另一方面看，这个 API 有多么不容易改动？
- 删除模块的一部分、完全删除，或用其他逻辑代替它是不是很容易？
- 如果模块之间的依赖度很高，那代码年代越久远，改版次数越多，代码量越大，修改就越困难。

浏览器实现的功能只是原生 JavaScript 模块的一点皮毛。在撰写本书时，有的浏览器已经实现了 `import` 和 `export` 语句。有的浏览器已经实现了 `<script type='module'>`，通过指定 `module` 脚本类型来使用模块。模块加载器规范还未最终制定完成，其最新进展参见 https://github.com/whatwg/loader#implementation-status。

在此期间，Node.js 发布的新版本还没有包含 JavaScript 模块系统的实现。考虑到 JavaScript 生态系统中的工具都依赖 Node，到底能实现多大程度的兼容还说不清楚。实现迟迟不能推出的原因是，目前还无法确定一个文件是 CommonJS 模块还是 ESM 模块。根据文件中至少存在一个 `import` 或 `export` 语句来判断它是否为 ESM 模块的提案最终被废弃了。目前的做法是准备为 ESM 模块专门引入一个新文件扩展名。鉴于运行 Node.js 的平台及使用场景具有多样性，这里要考虑的细节非常庞杂。得到一个优雅、完美、正确的方案是非常难的。

下一章将讨论如何有效地使用这些新的语言特性和语法。

第 9 章

实用建议

JavaScript 是一门不断发展的语言。多年来，JavaScript 的发展节奏有快有慢，随着 ES5 的推出，其发展进入了高速阶段。到目前为止，本书介绍了 ES6 中引入的几十种语言特性和语法变化，以及后来在 ES2016 和 ES2017 中发布的一些语法特性和语法变更。

将所有这些新特性与我们现有的 ES5 知识进行协调似乎是一项艰巨的任务：我们应该使用哪些特性以及如何使用呢？本章旨在帮助我们在使用 ES6 新特性时做出更合理的选择。

我们将探究不同的特性，学习这些特性的用法，并探讨何种情况下使用旧特性更为适合。我们来一个接一个地看。

9.1　变量声明

在开发软件时，我们的大部分时间都花在了阅读代码上，而不是编写代码。ES6 提供了 `let` 和 `const` 作为变量声明的新方式，并且这些语句中的部分值可以用于标记变量的使用方式。在阅读一段代码时，你可以从这些标记中得到线索，以便更好地理解作者的意图。此类线索对于缩短理解代码的时间至关重要，因此我们应尽可能多地使用它们。

由于需要遵循临时死区原则，`let` 语句表明变量不能在其声明前使用。这不是惯例，而是事实：如果我们在声明语句前尝试访问变量，则程序将失败。这些语句是块级作用域的，而不是函数作用域的。`let` 语句可以帮助我们在阅读更少代码的情况下完全掌握一个变量的用法。

`const` 语句也是块级作用域的，同样遵循临时死区原则。`const` 绑定只能在声明时赋值，因此更容易掌握。

注意，尽管上述规则表示变量绑定无法更改，但这并不意味着变量值本身永远不可变。绑定对象引用的 const 变量不能更改绑定，但是这个被引用的对象是可变的。

除了与 let 相同的功能，const 关键字还表示变量绑定不能被重新赋值。这是一个更为强烈的信号。你知道值是什么；你知道由于块级作用域，绑定的变量不能在包含它的块外被访问；你也知道由于临时死区原则，绑定在声明之前从未被访问。

你可以知道所有的一切并且无须搜寻变量的其他引用，只因为该变量是通过 const 声明的。

let 和 const 声明语句提供的约束条件可以使得代码更易理解。因此，编写代码的过程中应该尽可能多地尝试添加这些约束条件。一段代码的声明性约束越多，日后人们阅读、解析和理解这段代码就越容易、越迅速。

显然，const 声明比 var 声明的规则更多，包括块级作用域、临时死区原则、声明时赋值以及禁止重新赋值，而 var 声明只规定了函数作用域。条条框框太多也许会导致作茧自缚，我们最好从复杂性来权衡这些规则：规则增加还是减少了复杂性？在 const 中，块级作用域意味着作用域的范围比函数作用域更窄；临时死区原则意味着无须追溯声明前的代码来查找变量用法；赋值规则确保绑定将始终保留相同的引用。

限制性语句越多，代码就会越简单。当我们为语句的含义添加约束时，代码会变得更加可预测。这通常是静态类型程序比动态类型程序更易阅读的原因之一。静态类型会很大程度地限制程序编写者及程序的解析，从而使得代码更易理解。

综上所述，建议你尽可能多地使用 const，因为这种声明允许我们尽可能少地思考。

```
if (condition) {
  // 在符合条件之前不能访问isReady
  const isReady = true
  // isReady不能被重新赋值
}
// 在块级作用域之外不能访问isReady
```

考虑到有些变量后续要被重新赋值，我们会选择使用 let 进行变量声明，而非 const。let 语句与 const 语句具有相同的优点，但 let 语句声明的变量可以重复赋值。let 语句可用于实现计数器、改变布尔值或推迟初始化。

思考以下示例，函数输入是表示兆字节（MB）的数字，返回值是一个字符串，如 1.2GB。此处，我们使用 let，因为一旦满足条件，变量值就需要改变。

```
function prettySize(input) {
  let value = input
  let unit = 'MB'
  if (value >= 1024) {
    value /= 1024
    unit = 'GB'
```

```
  }
  if (value >= 1024) {
    value /= 1024
    unit = 'TB'
  }
  return `${ value.toFixed(1) } ${ unit }`
}
```

要想支持拍字节（PB）级单位，就需要在 `return` 语句前引入一个新的 `if` 分支。

```
if (value >= 1024) {
  value /= 1024
  unit = 'PB'
}
```

如果想让 `prettySize` 更轻松地扩展新的单位，我们可以考虑实现一个 `toLargestUnit` 函数来计算任何给定 `input` 的 `unit` 和 `value` 及其当前的单位。然后我们可以在 `prettySize` 中使用 `toLargestUnit` 来返回格式化的字符串。

以下代码实现了这样一个功能。它依赖于支持的 `units` 数组，而不是为每个单位都创建新的分支。当输入 `value` 不小于 `1024` 且存在更大的单位时，我们将输入的数值除以 `1024` 并移到下一个单位。接着我们对更新后的值调用 `toLargestUnit`，它将继续递归地减小 `value`，直到它足够小或者我们达到最大单位。

```
function toLargestUnit(value, unit = 'MB') {
  const units = ['MB', 'GB', 'TB']
  const i = units.indexOf(unit)
  const nextUnit = units[i + 1]
  if (value >= 1024 && nextUnit) {
    return toLargestUnit(value / 1024, nextUnit)
  }
  return { value, unit }
}
```

过去，引入对 PB 的支持需要涉及新的 `if` 分支和重复的逻辑，现在只需要在 `units` 数组末尾添加 `'PB'` 字符串。

`prettySize` 函数只需要专注于解决字符串的显示问题，因为它可以将计算的逻辑转移到 `toLargestUnit` 函数中。这种专注分离也有助于增加代码的可读性。

```
function prettySize(input) {
  const { value, unit } = toLargestUnit(input)
  return `${ value.toFixed(1) } ${ unit }`
}
```

当代码中存在需要被重新赋值的变量时，我们应该花几分钟思考一下，是否有更好的模式来解决相同的问题，同时可以避免重新赋值。这种模式并非一定有，但大部分时候是存在的。

一旦想到不同的解决方案，你就可以将其与之前的方案进行比较。确保代码可读性得到改善，并且实现仍然正确。这里进行单元测试可能是有用的，因为它们可以确保你不会犯重复的错误。如果重构的代码在可读性或可扩展性方面看起来更差，那么可以考虑回到以前的解决方案。

思考以下示例，其中用数组连接来生成 `result` 数组。这里我们可以进行简单的修改，将 `let` 变成 `const`。

```
function makeCollection(size) {
  let result = []
  if (size > 0) {
    result = result.concat([1, 2])
  }
  if (size > 1) {
    result = result.concat([3, 4])
  }
  if (size > 2) {
    result = result.concat([5, 6])
  }
  return result
}
makeCollection(0) // <- []
makeCollection(1) // <- [1, 2]
makeCollection(2) // <- [1, 2, 3, 4]
makeCollection(3) // <- [1, 2, 3, 4, 5, 6]
```

我们可以用接受多个值的 `Array#push` 方法来替换重新赋值的操作。如果是一个动态列表，则可以使用扩展运算符来放入更多的 `...items`（即展开更多项）。

```
function makeCollection(size) {
  const result = []
  if (size > 0) {
    result.push(1, 2)
  }
  if (size > 1) {
    result.push(3, 4)
  }
  if (size > 2) {
    result.push(5, 6)
  }
  return result
}
makeCollection(0) // <- []
makeCollection(1) // <- [1, 2]
makeCollection(2) // <- [1, 2, 3, 4]
makeCollection(3) // <- [1, 2, 3, 4, 5, 6]
```

当确实需要使用 `Array#concat` 时，你可以使用 `[...result, 1, 2]` 来缩短代码。

最后我们来看看重构。有时一些大型函数中可能会出现以下这种代码。

```
let completionText = 'in progress'
if (completionPercent >= 85) {
  completionText = 'almost done'
} else if (completionPercent >= 70) {
  completionText = 'reticulating splines'
}
```

在这些情况下，将逻辑提取为纯函数是有意义的。这样不仅可以减少在大型函数顶部出现复杂的初始化代码，还可以将计算文本的逻辑汇总到一个地方。

以下代码展示了如何将计算文本的逻辑抽取为一个函数。这样我们便可以将 `getCompletionText` 从原来的函数中分离，使代码更为线性化，从而提升可读性。

```
const completionText = getCompletionText(completionPercent)
// ...
function getCompletionText(progress) {
  if (progress >= 85) {
    return 'almost done'
  }
  if (progress >= 70) {
    return 'reticulating splines'
  }
  return 'in progress'
}
```

9.2 模板字面量

长久以来，由于格式化字符串不是原生方法，JavaScript 开发者需要借助第三方库来实现。创建多行字符串也很麻烦，比如对单引号或双引号进行转义，这取决于你使用的是哪种引用样式。如今，模板字面量的产生简化了这些操作。

模板字面量允许嵌入表达式，这样就可以在字符串中实现变量、函数调用或任何 JavaScript 表达式的内联操作，而无须依赖任何连接符。

```
'Hello, ' + name + '!' // 以前的做法
`Hello, ${ name }!` // 使用模板字符串的做法
```

以下代码是一段多行字符串，其中包含一个或多个数组连接、字符串连接及换行符 `\n`。这是 ES6 问世前书写 HTML 字符串的典型场景。

```
'<div>' `
  '<p>' `
    '<span>Hello</span>' `
    '<span>' + name + '</span>' `
    '<span>!</span>' `
  '</p>' `
'</div>'
```

通过使用模板字面量，我们可以避免所有不必要的引号和连接，并专注于内容。支持嵌入表达式很有用，这一功能也使得多行字符串成为模板字面量最常用的场景之一。

```
`<div>
  <p>
    <span>Hello</span>
    <span>${ name }</span>
    <span>!</span>
  </p>
</div>`
```

当一个字符串包含引用时，' 和 " 要比 ` 更为有用。对于大多数的英语句式，单引号和双引号的使用频率要比反引号高得多，因此反引号可以减少转义的使用[1]。

```
'Alfred\'s cat suit is "slick".'
"Alfred's cat suit is \"slick\"."
`Alfred's cat suit is "slick".`
```

正如第 2 章中所说，还有一些其他的特性，比如标签字符串，它可以使嵌入表达式更为简洁。虽然这一功能很有用，但它还是不如多行字符串、嵌入表达式及减少转义好用。

综合以上特性可知，模板字面量比单引号或双引号字符串更适合作为默认字符串样式。与此同时，仍然存在一些问题。接下来我们将讨论并解决每个问题。你可以自行决定如何使用。

当需要在字符串内嵌入表达式时，使用模板字面量比字符串连接更合适。或许大家都认同这个观点，那么我们就从这个观点开始讨论。

性能通常是大家关心的问题之一。所有地方都使用模板字面量会降低应用的性能吗？当使用 Babel 等编译器时，模板字面量会转换为带引号的字符串，表达式会通过连接符穿插其间。

思考使用模板字面量的以下示例。

```
const suitKind = `cat`
console.log(`Alfred's ${ suitKind } suit is "slick".`)
// <- Alfred's cat suit is "slick".
```

Babel 等编译器可以用带引号的字符串将示例代码转换为如下代码。

```
const suitKind = 'cat'
console.log('Alfred\'s ' + suitKind + ' suit is "slick".')
// <- Alfred's cat suit is "slick".
```

就可读性而言，我们知道嵌入表达式比带引号的字符串拼接更好，但为了最大程度地兼容

注 1：排版爱好者可能会说排版中不应该使用直引号，而应该使用无须转义的曲引号，即“”和‘’。诚然这是正确的，但由于直引号更容易输入，因此编写代码时习惯这样使用。此外，排版美化的工作如今已经交给第三方库或 Markdown 等编译器了，事情变得简单很多。

浏览器，编译器会将这些嵌入表达式转换为带引号的字符串拼接。

当模板字面量中只有 `suitKind` 变量，没有插值、换行符或标签时，编译器会简单地将其变成一个普通的带引号的字符串。

如果不进行从模板字面量到带引号字符串的转换，那么我们最好设法优化编译器，以确保准确编译的情况下将延迟降到最低。

另一个经常被指出的问题是语法。在撰写本书时，我们不能在 JSON、对象键、`import` 声明及严格模式指令中使用反引号字符串。

以下代码中的第一条语句表明，序列化的 JSON 对象不能使用反引号表示字符串。从第二行可以看出，我们可以用模板字面量声明一个对象，然后将该对象序列化为 JSON。在调用 `JSON.stringify` 时，模板字面量已经转译成了带引号的字符串。

```
JSON.parse('{ "payload": `message` }')
// <- SyntaxError
JSON.stringify({ payload: `message` })
// <- '{"payload":"message"}'
```

模板字面量在对象的键中并无用武之地，这样写会导致语法错误。

```
const alfred = { `suit kind`: `cat` }
```

对象属性名接受值类型，这些值类型会转换为纯字符串，但模板字面量不是值类型，因此无法用作属性名。

或许你会想到第 2 章中介绍过的 ES6 的可计算属性名，如下所示。在可计算属性名中，我们可以使用任何想要生成所需属性键的表达式，其中包括模板字面量。

```
const alfred = { [`suit kind`]: `cat` }
```

但是这样的写法过于冗长，不推荐使用。此时，常规的带引号的字符串才是最佳实践。

记住，“模板字面量是最佳选择”这一规则并不永远正确，当某条规则并不适用于你的使用场景、使用习惯或代码结构时，需要抱着开放的心态做出自己的判断，必要时可以打破它。规则是人制定的，同样的规则也许适合你，但并不适合别人。这就是现代代码检查器中的每条规则都是可选的原因，我们必须遵从规则，但这些规则并非适用于所有项目。

或许有一天我们会发明一种写法，可计算属性名不需要写方括号就能使用模板字面量，这样进行内嵌字符串时也可以少码点字。但短时间内，以下这种写法是有语法错误的。

```
const brand = `Porsche`
const car = {
  `wheels`: 4,
  `has fuel`: true,
```

```
  `is ${ brand }`: `you wish`
}
```

导入模块时使用模板字面量同样会导致语法错误。我们希望可以在代码中使用以下写法来进行模块导入，但事实上并做不到。

```
import { SayHello } from `./World`
```

严格模式指令必须是单引号或双引号字符串。在撰写本书时，并没有计划允许模板字面量用于 `'use strict'` 指令。虽然以下代码段不会导致语法错误，但它也不会启用严格模式。这是大量使用模板字面量时最需要注意的一点。

```
'use strict' // 启用严格模式
"use strict" // 启用严格模式
`use strict` // 严格模式未生效
```

将现有代码中带引号的字符串都换成模板字面量这一做法目前存在争议，这样做很容易产生错误，因此最好在开发新功能或修复 bug 时渐进式地进行替换。

好在我们有 `eslint` 的支持，正如第 1 章中所讨论的那样。要想将代码的默认字符串样式切换为反引号，我们可以设置一个类似以下代码中的 .eslintrc.json 配置。注意，引用（`quotes`）需要使用反引号，否则 `eslint` 将报错。

```
{
  "env": {
    "es6": true
  },
  "extends": "eslint:recommended",
  "rules": {
    "quotes": ["error", "backtick"]
  }
}
```

基于此，我们可以在 package.json 中添加一个 `lint` 脚本，如下所示。其中 `--fix` 修饰符可以自动更正代码中的样式错误，比如，引用时使用单引号而非反引号。

```
{
  "scripts": {
    "lint": "eslint --fix ."
  }
}
```

在命令行中执行以下命令就可以让我们的代码默认使用反引号了。

```
» npm run lint
```

总之，使用模板字面量时需要权衡一下。你可以尝试采用反引号优先的方法并评估其优点。与惯例和配置相比，使用方便更重要。

9.3　简写及对象解构

第 1 章中介绍了简写的概念。当想要引入一个属性，同时作用域内存在同名绑定时，我们可以使用简写来避免重复书写。

```
const unitPrice = 1.25
const tomato = {
  name: 'Tomato',
  color: 'red',
  unitPrice
}
```

此特性在函数和信息隐藏中特别有用。以下示例用对象解构从一个商品中获取几条信息，并返回一个包含商品总价的模型。

```
function getGroceryModel({ name, unitPrice }, units) {
  return {
    name,
    unitPrice,
    units,
    totalPrice: unitPrice * units
  }
}
getGroceryModel(tomato, 4)
/*
{
  name: 'Tomato',
  unitPrice: 1.25,
  units: 4,
  totalPrice: 5
}
*/
```

以上代码将简写与对象解构搭配在一起使用，这种写法很好。如果将解构看作一种从对象中抽取属性的方式，那么反过来，简写就可以被看作一种向对象添加属性的方式。在以下示例中，知道客户购买的数量后，我们就可以用 `getGroceryModel` 函数来获取商品的 `totalPrice`。

```
const { totalPrice } = getGroceryModel(tomato, 4)
```

虽然刚开始接触时觉得这种做法有悖于直觉，但在函数参数中应用解构会很方便并且可预期，因为我们知道 `getGroceryModel` 的第一个参数一定包含 `name` 和 `unitPrice` 属性。

```
function getGroceryModel({ name, unitPrice }, units) {
  return {
    name,
    unitPrice,
    units,
    totalPrice: unitPrice * units
  }
}
```

如果解构函数的输出，则可以让读者立即明白输出的哪些内容是作者所关心的。在以下代码中，我们只需要产品名和总价，因此可以解构输出值来获取这两个值。

```
const { name, totalPrice } = getGroceryModel(tomato, 4)
```

与上面代码做法不同，以下代码没有使用解构，而是直接将输出的对象放入 model 中。很奇妙的是，这点区别导致以下代码传递的信息量变少了，我们需要查看后面的代码才能知道 model 中的哪些属性正在被使用。

```
const model = getGroceryModel(tomato, 4)
```

当使用同一对象的多个属性时，解构还可以帮助避免重复引用宿主对象。

```
const summary = `${ model.units }x ${ model.name }
($${ model.unitPrice }) = $${ model.totalPrice }`
// <- '4x Tomato ($1.25) = $5'
```

尽管解构可以避免使用变量时重复引用宿主对象，但解构声明中需要重复书写属性名，这也产生了一定的开销，需要对此进行权衡。

```
const { name, units, unitPrice, totalPrice } = model
const summary = `${ units }x ${ name } ($${ unitPrice }) =
$${ totalPrice }`
```

如果需要多次引用同一属性，那么我们最好使用对象解构，以避免重复引用宿主对象。

但只用到一个属性且只使用一次时，为简单起见，显然应该避免使用解构。

```
const { name } = model
const summary = `This is a ${ name } summary`
```

在 summary 中直接引用 model.name 更简洁。

```
const summary = `This is a ${ model.name } summary`
```

当需要使用两个属性（或者引用两次同一属性）时，情况就不一样了。

```
const summary = `This is a summary for ${ model.units }x
${ model.name }`
```

这种情况下使用解构更好。解构不仅减少了 summary 声明语句中的字符数，还可以很清晰地表示出我们需要使用 model 中的哪些属性。

```
const { name, units } = model
const summary = `This is a summary for ${ units }x ${ name }`
```

如果我们引用同一属性两次，那么也应该使用解构。在以下示例中，相较于不使用解构，我们少了一个 model 引用并多了一个 name 引用。虽然解构可以明确地表明 name 的预期用

法，这很有用，但使用或者不使用解构在本例中都可以接受。

```
const { name } = model
const summary = `This is a ${ name } summary`
const description = `${ name } is a grocery item`
```

解构有助于减少对宿主对象的引用，但同时会带来声明时的无谓重复，我们需要根据引用属性的数量来决定是否使用它。总之，虽然解构这一特性很棒，但它并不总会改善代码的可读性，特别是无须减少宿主对象引用时，我们更应该辩证地使用它。

9.4 剩余参数和扩展运算符

正则表达式的匹配结果通常表示为数组，其中匹配结果在数组的第一位，而每个捕获组放置在数组的后续元素中。通常来说，我们对某些特定的捕获感兴趣，比如“第一个捕获”。

在以下代码中，我们使用解构将匹配结果中的两个捕获组分别赋给 integer 和 fractional 变量，这样就不需要依靠索引来引用相应的捕获组了。

```
function getNumberParts(number) {
  const rnumber = /(\d+)\.(\d+)/
  const matches = number.match(rnumber)
  if (matches === null) {
    return null
  }
  const [ , integer, fractional] = number.match(rnumber)
  return { integer, fractional }
}
getNumberParts('1234.56')
// <- { integer: '1234', fractional: '56' }
```

作为解构 .match 结果的一部分，我们可以用扩展运算符来获取每个捕获组。

```
function getNumberParts(number) {
  const rnumber = /(\d+)\.(\d+)/
  const matches = number.match(rnumber)
  if (matches === null) {
    return null
  }
  const [ , ...captures] = number.match(rnumber)
  return captures
}
getNumberParts('1234.56')
// <- ['1234', '56']
```

需要拼接数组时，通常使用 .concat 来创建一个新的数组。现在，使用扩展运算符可以清晰地表明需要创建一个由输入数组组成的新集合，并且添加元素的这种声明式方式更符合思维习惯，从而改善了代码的可读性。

```
administrators.concat(moderators)
[...administrators, ...moderators]
[...administrators, ...moderators, bob]
```

同样，3.3.1 节中介绍的对象扩展特性[2]支持将已有对象合并到新对象。以下代码创建了一个新对象，它将基础的 defaults 对象、用户提供的 options 对象和一些重要的重载属性合并在一起，后面的属性会覆盖前面的属性。

```
Object.assign({}, defaults, options, { important: true })
```

上述功能可以用声明式的对象扩展运算符来实现。我们使用对象字面量，并放入 important 属性、扩展后的 defaults 对象及 options 对象。避免使用 Object.assign 方法可以显著地提升代码的可读性，我们甚至可以将 important 属性内联到对象字面量声明中。

```
{
  ...defaults,
  ...options,
  important: true
}
```

能够将对象扩展可视化为 Object.assign 有助于我们更深刻地理解该特征的工作原理。以下示例用对象字面量替换了 defaults 变量和 options 变量。对每个属性来说，对象扩展和 Object.assign 的操作本质上相同，因此我们可以看到 speed 值为 3 的原因是 options 字面量进行了属性覆盖，即使 options 字面量试图覆盖，但 important 属性值仍然是 true，这是由优先权的关系决定的。

```
{
  ...{ // defaults
    speed: 1,
    type: 'sports'
  },
  ...{ // options
    speed: 3,
    important: false
  },
  important: true
}
```

在处理不可变结构时，对象扩展会很有用，我们应该创建新对象，而不是编辑现有对象。以下代码中有一个 player 对象和一个施展治疗法术的函数调用，该函数返回一个新的、更健康的 player 对象。

```
const player = {
  strength: 4,
  luck: 2,
```

注 2：撰写本书时处于 ECMAScript 标准开发过程的阶段 3。

```
  mana: 80,
  health: 10
}
castHealingSpell(player) // 消耗40点法力，获得110点生命值
```

以下代码展示了 castHealingSpell 方法的实现过程，我们在不改变原始 player 参数的情况下创建了一个新的 player 对象。原始 player 对象中的每个属性都被复制了，我们可以根据需要更新各个属性。

```
const castHealingSpell = player => ({
  ...player,
  mana: player.mana - 40,
  health: player.health + 110
})
```

正如第 3 章中所阐述的那样，我们可以使用对象的剩余属性来解构对象。除了枚举未知属性等用途，对象的剩余属性还可以用于创建对象的浅副本。

以下代码展示了三种最简单的在 JavaScript 中创建对象浅副本的方法。第一种方法是使用 Object.assign，将 source 对象的每个属性赋给一个空对象，然后返回该对象。第二种方法是使用对象扩展运算符，这相当于使用 Object.assign，但实现方式更为优雅。最后一种方法依赖于解构剩余参数。

```
const copy = Object.assign({}, source)
const copy = { ...source }
const { ...copy } = source
```

有时我们需要创建一个对象的副本，同时在该副本中省略一些属性。例如，我们可能希望创建 person 对象的副本，同时忽略其中的 name 属性，只保留元数据。

用普通的 JavaScript 实现这种方法的方式是，使用剩余参数来解构 name 属性，并将其他属性放置在 metadata 对象中。我们已经有效地将无须关心的 name 属性从 metadata 对象中“移除”了，因此 metadata 对象包含了 person 对象中除 name 属性外的所有其他属性。

```
const { name, ...metadata } = person
```

以下代码将 people 数组映射为一组 person 模型，其中去除了个人身份信息（如姓名和身份证号），同时将其他所有内容放入 person 这个剩余参数中。

```
people.map(({ name, ssn, ...person }) => person)
```

9.5　函数偏好

在 ES6 问世前，JavaScript 已经提供了许多函数声明方法。

函数声明是最重要的一种 JavaScript 函数。函数声明未被强调意味着我们可以依据代码可

读性给函数声明排序，而无须担心实际使用顺序。

以下代码展示了三个函数声明的排列顺序，这种顺序有助于线性阅读。

```
printSum(2, 3)
function printSum(x, y) {
  return print(sum(x, y))
}
function sum(x, y) {
  return x + y
}
function print(message) {
  console.log(`printing: ${ message }`)
}
```

相反，函数表达式必须先分配给一个变量，然后才能执行它们。继续前面的示例，这意味着我们必须在实际使用函数前先对其进行声明。

以下代码使用函数表达式。注意，如果我们将 `printSum` 函数调用放在非末尾的其他位置，那么代码会因为变量尚未初始化而失败。

```
var printSum = function (x, y) {
  return print(sum(x, y))
}
var sum = function (x, y) {
  return x + y
}
// 因为print未定义，所以执行printSum()会失败
var print = function (message) {
  console.log(`printing: ${ message }`)
}
printSum(2, 3)
```

因此，将函数表达式排序为后进先出的栈可能会更好：将最后调用的函数放在第一个，将倒数第二调用的函数放在第二个，依此类推。重新排列的代码如下所示。

```
var sum = function (x, y) {
  return x + y
}
var print = function (message) {
  console.log(`printing: ${ message }`)
}
var printSum = function (x, y) {
  return print(sum(x, y))
}
printSum(2, 3)
```

虽然这段代码有点难以读懂，但显然，将函数表达式赋给 `printSum` 变量前不能调用它。在上一段代码中，这点并不明显，因为我们没有遵循后进先出的规则。对于绝大多数代码来说，这足以让我们更倾向于函数声明。

函数表达式可以有一个用于递归的名称，但该名称在外部作用域中不可访问。以下代码中的具名函数表达式 sum 赋给了变量 sumMany。sum 引用可用于内部作用域中的递归，但试图从外部作用域使用该引用会出现错误。

```
var sumMany = function sum(accumulator = 0, ...values) {
  if (values.length === 0) {
    return accumulator
  }
  const [value, ...rest] = values
  return sum(accumulator + value, ...rest)
}
console.log(sumMany(0, 1, 2, 3, 4))
// <- 10
console.log(sum())
// <- ReferenceError: sum is not defined
```

2.2 节中介绍的箭头函数与函数表达式类似，它舍弃了 function 关键字，使得语法更为简洁。在箭头函数中，当只有一个参数，且该参数既没有被解构，又不是剩余参数时，可以不使用圆括号将其括起来。箭头函数可以隐式地返回任何有效的 JavaScript 表达式，而无须声明块语句。

以下代码展示了箭头函数的用法。第一个箭头函数在块语句中显式地返回一个表达式；第二个则隐式地返回表达式；第三个省略了唯一参数周围的圆括号；第四个使用块语句，但并不返回值。

```
const sum = (x, y) => { return x + y }
const multiply = (x, y) => x * y
const double = x => x * 2
const print = x => { console.log(x) }
```

箭头函数可以用微小的表达式返回数组。以下代码中的第一个示例隐式地返回包含两个元素的数组，第二个示例舍弃了第一个参数并将其余的参数放在剩余参数 items 中返回。

```
const makeArray = (first, second) => [first, second]
const makeSlice = (discarded, ...items) => items
```

隐式地返回一个对象字面量有点棘手，因为很难将其与同样使用大括号包裹的块语句进行区分。我们必须在对象字面量周围添加圆括号，将其转化为一个对象表达式。这样处理足以消除歧义，并告诉 JavaScript 解析器正在处理的是对象字面量。

思考以下示例，我们隐式地返回了一个对象表达式。如果没有圆括号，解析器会将代码解析为包含标签和字面量表达式 'Nico' 的块语句。

```
const getPerson = name => ({
  name: 'Nico'
})
```

由于语法限制，我们无法为箭头函数显式地命名。但如果将箭头函数声明放在变量或属性的右侧，则该变量名或属性名就成为了箭头函数的名字。

使用前需要给箭头函数表达式赋值，因此会遇到与常规函数表达式相同的排序问题。此外，由于无法命名，我们必须将箭头函数绑定到一个变量，这样才能在递归时引用它们。

默认情况下，我们应该首选使用函数声明，其在排序、引用和执行方面限制较少，因此可以获得更好的代码可读性和可维护性。在未来的重构中，也就不必担心因排列顺序错误导致的依赖链断裂或必须遵循后进先出的限制。

再说箭头函数，它是一种以简短形式声明函数的简洁而强大的方式。函数越小，就越适合使用箭头函数，这有助于避免耗费精力在形式上，从而只专注于函数本身。随着函数越来越大，由于前文提到的顺序和命名问题，箭头函数的写法就没那么吸引人了。

此外，在声明测试用例、`new Promise()` 和 `setTimeout` 的参数函数或数组映射函数等异步函数时，箭头函数也极其有用。

思考以下示例，我们使用非阻塞的 `wait` Promise 来实现 5 秒后打印一句话。`wait` 函数接受一个以毫秒为单位的 `delay` 参数，并返回一个 `Promise`，该 `Promise` 使用 `setTimeout` 在指定时间后进行解决。

```
wait(5000).then(function () {
  console.log('waited 5 seconds!')
})

function wait(delay) {
  return new Promise(function (resolve) {
    setTimeout(function () {
      resolve()
    }, delay)
  })
}
```

当用箭头函数重写时，我们应该保持 `wait` 函数的声明方式，这样就不用将它提升到作用域的顶部了。之后我们将其他函数都换成箭头函数，以去除为标记函数功能所使用的 `function` 关键字。

以下是用箭头函数重写后的代码。重构删除了 `function` 关键字，使得 `wait` 函数的 `delay` 参数和 `setTimeout` 的第二个参数间的关系更易理解。

```
wait(5000).then(
  () => console.log('waited 5 seconds!')
)

function wait(delay) {
  return new Promise(resolve =>
```

```
    setTimeout(() => resolve(), delay)
  )
}
```

使用箭头函数的另一大优势在于它的词法作用域，其内部不会修改 this 和 arguments 的指向。如果目前的代码需要将 this 赋给一个临时变量（通常命名为 self、context 或者 _this）才能在其内部函数中获取，那么我们可以换成箭头函数的写法。我们来看看以下示例。

```
const pistol = {
  caliber: 50,
  trigger() {
    const self = this
    setTimeout(function () {
      console.log(`Fired caliber ${ self.caliber } pistol`)
    }, 1000)
  }
}
pistol.trigger()
```

如果试图在前面示例中直接使用 this，那么我们会得到 undefined。但使用箭头函数就可以避免使用临时变量 self。我们不仅删除了 function 关键字，还得益于词法作用域，可以不用因为语法限制而改变写法。

```
const pistol = {
  caliber: 50,
  trigger() {
    setTimeout(() => {
      console.log(`Fired caliber ${ self.caliber } pistol`)
    }, 1000)
  }
}
pistol.trigger()
```

从经验出发，默认情况下使用函数声明。但如果该函数不需要有意义的名称、不包含大段代码或涉及递归，则可以考虑使用箭头函数。

9.6　类和代理

大多数的现代编程语言都有类的概念。JavaScript 类是基于原型继承的语法糖。类可以使原型更符合思维习惯，并且更易于工具进行静态分析。

在使用原型的方式书写时，同名函数就是构造函数本身，而声明实例方法则需要套用固定的代码模式，如下所示。

```
function Player() {
  this.health = 5
}
Player.prototype.damage = function () {
```

```
  this.health--
}
Player.prototype.attack = function (player) {
  player.damage()
}
```

相比之下，类将 constructor 规范化为一个实例方法，因此创建每个实例时都会调用这个构造函数。同时，方法被内置到 class 字面量中，并采用与对象字面量中的方法一致的语法。

```
class Player {
  constructor() {
    this.health = 5
  }
  damage() {
    this.health--
  }
  attack(player) {
    player.damage()
  }
}
```

将所有的实例方法全部放在对象字面量内部，可以确保类声明不会散落分布在不同的文件中，因而开发者可以在一个位置中描述完整的 API。

相较于将 static 方法动态注入类，将其声明为 class 字面量的一部分同样有助于将 API 集中化。API 集中化有助于提高代码的可读性，开发者学习 Player API 时需要阅读的代码更少。同时，一旦我们约定在 class 字面量中声明实例和静态方法，编码人员就不用浪费时间在别处寻找动态方法。同样，我们也可以在 class 字面量中定义 getter 和 setter。

```
class Player {
  constructor() {
    Player.heal(this)
  }
  damage() {
    this.health--
  }
  attack(player) {
    player.damage()
  }
  get alive() {
    return this.health > 0
  }
  static heal(player) {
    player.health = 5
  }
}
```

类还提供了 extends 这个基于原型继承的简单语法糖。使用 extends 比直接使用基于原型的解决方案更方便，因为我们无须依赖于第三方库或采用其他动态方法来实现类的继承。

```
class GameMaster extends Player {
  constructor(...rest) {
    super(...rest)
    this.health = Infinity
  }
  kill(player) {
    while (player.alive) {
      player.damage()
    }
  }
}
```

类可以不依赖 `<iframe>` 或浅副本就使用同样的语法对内置的原生对象（如 `Array` 和 `Date`）进行扩展。以下代码中的 `List` 类对 `Array` 进行扩展，从而消除构建 `Array` 时单参数重载的影响，同时将自定义的方法放置到 `Array` 的原型链上。

```
class List extends Array {
  constructor(...items) {
    super()
    this.push(...items)
  }
  get first() {
    return this[0]
  }
  get last() {
    return this[this.length - 1]
  }
}
const number = new List(2)
console.log(number.first)
// <- 2
const items = new List('a', 'few', 'examples')
console.log(items.last)
// <- 'examples'
```

与原型链相比，JavaScript 类没那么冗长，因此类这个语法糖比原型继承更受欢迎。使用 JavaScript 类是否有好处却是见仁见智的。尽管类由于其先进的语法受到了更多关注，但仅凭语法糖不足以让它得到广泛的应用。

静态类型的语言通常提供并强制使用类[3]。相比之下，由于 JavaScript 具有高度动态性，类并不是强制性的。几乎所有需要使用类的场景都可以用普通对象来解决。

普通对象比类更简单，其初始化的唯一方式是声明，无须使用特殊的构造函数方法。它们很容易通过 JSON 进行序列化，并且更具互操作性。继承很少使用正确的抽象，但需要时我们可以切换到类，或使用普通对象和 `Object.create`。

代理使得许多以前不可用的用例成为可能，但需要慎用。涉及 `Proxy` 对象的解决方案也可以使用普通对象和函数来实现，而无须诉诸神奇的 `Proxy` 对象。

注 3：大多数的函数式编程语言都应该是例外。

确实可能存在必须使用 Proxy 的情况，特别是涉及开发环境中的开发者工具时，其中开发者工具的代码库中隐藏了高度的代码内省及复杂性。在应用级代码中使用 Proxy 可以很轻松地避免这类问题，从而减少费解的代码。

代码的目的越明确，其可读性就越好。声明式代码是可读的，明确要做的事情使得代码更为清晰。相比之下，在对象之上使用间接层（如 Proxy）会产生非常复杂且难以推断的访问规则。这并不是说涉及 Proxy 的解决方案就难以理解，而是需要阅读和仔细考虑更多代码才能充分理解代理层行为方式的细微差别。

如果想要使用代理，那么对象可能不是我们解决问题所适用的工具。不要直接进入 Proxy 间接层，而是要考虑一个简单的函数是否提供了足够的间接性，并且不会导致对象的行为方式与 JavaScript 中普通对象的行为方式不一致。

因此，尽量选用乏味的、静态的、声明式的代码，而不是聪明的、优雅的抽象。乏味的代码可能比抽象有更多的重复，但它也会更简单、更容易理解，并且短期内肯定会更安全。

抽象是需要代价的。一旦抽象到位，往往很难回头并消除它。如果抽象创建得太早，它可能不会涵盖所有的常见用例，我们也可能最终不得不分别处理那些特殊情况。

选择乏味的代码时会逐渐自然地总结出某种模式。一旦出现这种模式，我们就可以决定是否需要进行抽象并适量重构我们的代码。一旦有两三个功能相当的代码段，我们就已经抽象出了抽象概念，经过时间考验的抽象可能会涵盖更多的用例。

9.7 异步代码流

第 4 章探讨了管理异步操作复杂性的不同方式，以及如何使用它们。这些方式包含回调、事件、Promise、生成器、异步函数和异步迭代器、外部库和列表等。你现在应该已经熟悉了这些结构的工作方式，那么应该何时使用它们呢？

回调是最原始的解决方案。理解回调只需要一些基础的 JavaScript 知识，因此基于回调的代码易于阅读。一系列深度嵌套的异步操作可能会导致回调地狱，因此，在操作流涉及长依赖链的情况下，我们应该小心处理回调。

谈到回调，当我们有三个及以上相关任务需要异步执行时，可以使用 async[4] 这样的库来降低代码复杂性。这些库的另一个优点是对回调的微妙处理方式，这些库在处理复杂代码时很有用，比如那些综合了通过库抽象的复杂流和通过回调表达的简单代码流的代码。

事件是一种以异步等形式向代码流中引入可扩展性的简单方式。但是事件并不适合管理异步任务的复杂性。

注 4：你可以在 GitHub 上查找到的一个流行的流控制库。

以下示例表明，如果想要用事件处理异步任务，那么代码会变得很复杂。虽然一半的代码行用于定义代码流，但代码流还是很难理解。或许这意味着我们选错工具了。

```
const tracker = emitter()
tracker.on('started', multiply)
tracker.on('multiplied', print)
start(256, 512, 1024)
function start(...input) {
  const sum = input.reduce((a, b) => a + b, 0)
  tracker.emit('started', { sum, input })
}
function multiply({ sum, input }) {
  const message = `The sum of ${ input.join('`') } is ${ sum }`
  tracker.emit('multiplied', message)
}
function print(message) {
  console.log(message)
}
```

在 TC39 决定将它们引入核心 JavaScript 语言前，Promise 就已存在于用户库很久了。它们的作用与回调库类似，即提供了另一种编写异步代码流的方式。

Promise 比回调的使用成本更高一些，因为 Promise 链中包含更多的 Promise，所以它们很难与纯回调交错使用。同时，你也不希望将 Promise 与基于回调的代码搅在一起，这会使应用更复杂。对于任何给定的代码部分，选择一种形式并坚持很重要。只采取一种形式可以让代码更关注于处理的任务，而非实现机制。

并非使用 Promise 就一定不好，只是你需要了解一下成本。越来越多的网络平台将 Promise 作为基础构建块，它只会变得越来越好。Promise 是生成器、异步函数、异步迭代器和异步生成器的基础。使用 Promise 越多，应用就越具有协同性，虽然可以认为回调本质上已经具有协同性，但它们当然不能与异步函数和所有基于 Promise 的解决方案相比，并且这些方案现在是原生的 JavaScript 语言。

一旦选择 Promise，可供使用的工具方法与第三方库所提供的基于回调的解决方案在数量上就完全可以相提并论了。不同之处是，大多数情况下，Promise 不需要依赖任何库，因为它们现在是原生的 JavaScript 语言。

我们可以使用迭代器来简单描述可能有限的序列。此外，它们的异步对象可以用于描述需要外部处理（如 `GET` 请求）的序列，以产生元素。这些序列可以用 `for await..of` 循环来实现，从而避免了异步性质的复杂性。

迭代器是描述对象如何迭代生成序列的有效方式。当没有需要描述的对象时，生成器提供了描述独立序列的方式。实现一个迭代器是描述一个 `Movie` 对象应该如何迭代的理想方式，可能会使用 `Symbol.asyncIterator` 并获取电影中的每个演员及其所扮演的角色的信息。但是如果没有 `Movie` 对象的上下文，这样的迭代器作为生成器会更有意义。

生成器很有用的另一种情况是无限序列。思考以下迭代器，其中产生了无数的整数。

```
const integers = value => ({
  value,
  [Symbol.iterator]() {
    return {
      next: () => ({
        value: this.value++
      })
    }
  }
})
```

你可能还记得生成器本质上是可迭代的，这意味着它们遵循迭代器协议，因此我们无须提供迭代器。现在我们将可迭代 `integers` 对象与以下代码中的等价生成器函数进行比较。

```
function* integers(value = 0) {
  while (true) {
    yield value++
  }
}
```

生成器代码不仅更简短，而且更具可读性。由于 `while` 循环的缘故，它产生无限序列的事实变得非常明显。可迭代性要求我们理解序列是无限的，因为代码永远不会返回带有 `done: true` 标志的元素。设置初始 `value` 更自然，而且不涉及将对象包装在接受初始参数的函数中。

Promise 最初被誉为回调地狱的解药。拥有深度嵌套的异步串联流时，严重依赖 Promise 的程序可能会陷入回调陷阱。异步函数为这个问题提供了一个优雅的解决方案，我们可以使用 `await` 表达式来描述同样基于 Promise 的代码。

阅读以下代码。

```
Promise
  .resolve(2)
  .then(x => x * 2)
  .then(x => x * 2)
  .then(x => x * 2)
```

当我们使用一个 `await` 表达式时，其右边的表达式被强制转换为一个 Promise。当遇到 `await` 表达式时，异步函数将暂停执行，直到 Promise 被强制转换或以其他方式解决。当 Promise 完成时，异步函数继续执行，但如果 Promise 被拒绝，则拒绝结果将冒泡到由异步函数调用返回的 Promise 中，除非该拒绝被 `catch` 处理器捕获。

```
async function calculate() {
  let x = 2
  x = await x * 2
  x = await x * 2
  x = await x * 2
  return x
}
```

async/await 的优势是，它解决了 Promise 最大的问题，即无法轻易地将同步代码混合到工作流中。同时，异步函数允许使用 try/catch，而这是无法在回调中使用的。同时，通过在底层使用 Promise，async/await 始终能够与 Promise 保持一致，总是从每个异步函数中返回 Promise 并强制 await 表达式转换为 Promise。此外，在完成上述操作的同时，异步函数可以用一种类似同步的方式书写异步代码。

虽然使用 await 表达式可以优化并减少串联异步代码的复杂性，但用 async/ await 代替 Promise 时很难推断并发异步代码流。在达到 await 表达式之前，这可以通过 await Promise.all(tasks) 以及同时触发这些任务来缓解。但鉴于异步函数并未针对此用例进行优化，阅读此类代码可能会造成混淆，这一点需要注意。如果代码是高度并发的，则建议考虑使用基于回调的方法。

这又让我们辩证地思考问题，新的语言功能并不一定适合所有情况。遵守约定很重要，这不仅可以保持代码的一致性，还不用花大量时间来决定如何更好地展现一小段代码，但优雅的平衡也很重要。

当没有花时间搞清楚最适合自己的代码风格时，我们就是在冒险将所有问题都当成钉子，因为我们只有一把锤子。为手头的问题选择正确的工具比成为约定和硬性规则的卫道士更重要。

9.8　复杂性蠕变、抽象及约定

挑选正确的抽象很困难：我们想要通过引入隐藏在结构中的复杂性来降低代码流的复杂性。为此，异步函数借用了生成器的基础，生成器对象是可迭代的。异步迭代器使用了 Promise。迭代器使用了符号。Promise 使用了回调。

谈到可维护代码时，一致性是一个重要主题。一个应用可能主要使用回调或 Promise。就任意一个方法来说，回调和 Promise 都可以用来降低代码流的复杂性。当将它们混合使用时，我们需要确保不会引入上下文切换，以避免开发人员阅读不同的代码片段时需要用不同的思维方式来理解。

这就是约定存在的原因。“尽可能使用 Promise”等强有力的约定大大增加了代码的一致性。约定确保了代码库的可读性和可维护性。毕竟代码是用来传达信息的通信设备。此信息不仅与执行代码的计算机相关，而且对开发人员阅读代码、维护和改进应用也非常重要。

没有强有力的约定，沟通就会崩溃，开发人员就会很难理解程序如何工作，最终导致生产力下降。

开发人员的绝大多数时间都花在了阅读代码上。因此，我们应该投入更多精力来思考如何书写可读性强的代码。

版权声明

O'REILLY®

JavaScript之父Brendan Eich作序推荐

将JavaScript新特性融入简单易懂的示例中，助你大幅提升代码表达能力

本书从实际开发角度介绍ES6及后续更新版本特性，以循序渐进、通俗易懂的方式讲解各种复杂的技术，比如异步控制流、声明对象及函数的使用等，并从实践角度提供了许多建议，既能帮助广大前端开发者建立一个完整的知识体系，也能助其在工作中如虎添翼，开发出更好的Web应用。

- **JavaScript及其标准制定流程的变化**
- **ES6的根本性变化，包括箭头函数、解构、模板字面量等**
- **声明对象原型对应的类语法，以及新的基本数据类型：符号**
- **通过Promise、迭代器、生成器以及异步函数控制程序执行流**
- **如何创建对象映射和唯一值集合**
- **如何使用新的代理与反射特性**
- **数字、字符串、Unicode、正则表达式等内置API的改进**

“尼古拉斯写的东西特别实用……建议你好好读读，从中发现对自己有用的东西，进而真正拥抱JavaScript，致力于为所有人开发更好的Web应用。”

——**Brendan Eich**
JavaScript之父

“本书全面介绍了ES6新特性的语法和语义，有助于你大幅度提升代码的表达能力。作者把这些特性融入简单易懂的示例中，帮你快速上手。”

——**Kent C. Dodds**
PayPal前端工程师，TC39成员

尼古拉斯·贝瓦夸（Nicolás Bevacqua），知名JavaScript布道师，来自阿根廷的JavaScript编程高手，目前是Elastic公司用户界面工程师。另著有《JavaScript Web应用开发》一书。

封面设计：**Karen Montgomery** 张健

图灵社区：iTuring.cn
热线：(010)51095183转600

分类建议 计算机/程序设计/JavaScript

人民邮电出版社网址：www.ptpress.com.cn

O'Reilly Media, Inc.授权人民邮电出版社出版



ISBN 978-7-115-51040-2
定价：79.00元

看完了

如果您对本书内容有疑问，可发邮件至 contact@turingbook.com，会有编辑或作译者协助答疑。也可访问图灵社区，参与本书讨论。

如果是有关电子书的建议或问题，请联系专用客服邮箱:
ebook@turingbook.com。

在这可以找到我们:

微博 @图灵教育：好书、活动每日播报
微博 @图灵社区：电子书和好文章的消息
微博 @图灵新知：图灵教育的科普小组
微信 图灵访谈：ituring_interview，讲述码农精彩人生
微信 图灵教育：turingbooks